The Pragmatic Programmers

TURING 图灵程序设计丛书

Code in the Cloud

Programming Google AppEngine

云端代码

Google App Engine编程指南

[美] Mark C. Chu-Carroll 著

刘姝　管雪涛 译





人民邮电出版社
POSTS & TELECOM PRESS

- 实用的云计算编程开发教程
- 学习Google App Engine的入门佳作
- 让应用程序的代码从容驾驭浮云

Mark Chu-Carroll，谷歌软件工程师，从事软件开发将近20年。业余时，他还负责开发和管理Scientopia.org，并在该网站上发表数学博客Good Math/Bad Math，其博客地址为http://scientopia.org/blogs/goodmath。

刘姝博士，2010年毕业于北京大学，主要研究领域包括嵌入式实时操作系统、网络安全、云计算；参与了多项国家重大专项课题，先后发表论文十余篇。

管雪涛博士，2006年毕业于北京大学，目前在北京大学信息科学技术学院从事教学科研工作；主要研究方向包括操作系统原理、软硬件协同设计、虚拟化技术；在相关领域发表论文十余篇，并有9项发明专利获得授权。

TURING 图灵程序设计丛书

Code in the Cloud

Programming Google AppEngine

云端代码

Google App Engine编程指南

[美] Mark C. Chu-Carroll 著

刘姝　管雪涛 译



人 民 邮 电 出 版 社

北　京

图书在版编目（CIP）数据

云端代码 : Google App Engine编程指南 / (美) 卡罗尔 (Carroll,M.C.) 著 ; 刘姝, 管雪涛译. -- 北京 : 人民邮电出版社, 2013.1
（图灵程序设计丛书）
书名原文: Code in the Cloud: Programming Google AppEngine
ISBN 978-7-115-30199-4

Ⅰ. ①云… Ⅱ. ①卡… ②刘… ③管… Ⅲ. ①网页制作工具－程序设计－指南 Ⅳ. ①TP393.092-62

中国版本图书馆CIP数据核字(2012)第289214号

内 容 提 要

本书介绍了如何将应用程序构建为服务，如何使用 App Engine 管理持久化数据，如何构建可在用户浏览器上运行的、动态的、可交互的用户界面。如何管理 Web 应用的安全性，如何用 App Engine 与云端运行的其他服务交互。

本书适合云技术开发人员、Web 程序员阅读。

图灵程序设计丛书

云端代码：Google App Engine编程指南

◆ 著　[美] Mark C. Chu-Carroll
译　刘 姝　管雪涛
责任编辑　朱 巍
执行编辑　冯 郡

◆ 人民邮电出版社出版发行　北京市崇文区夕照寺街14号
邮编　100061　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　印刷

◆ 开本：800×1000 1/16
印张：14
字数：330千字　2013年1月第1版
印数：1－3 000册　2013年1月北京第1次印刷

著作权合同登记号　图字：01-2011-6037号

ISBN 978-7-115-30199-4

定价：45.00元

读者服务热线：(010)51095186转604　印装质量热线：(010)67129223

反盗版热线：(010)67171154

版 权 声 明

目　　录

第三部分 用Java进行Google App Engine编程

第四部分 高级Google App Engine编程

Part 1

第一部分

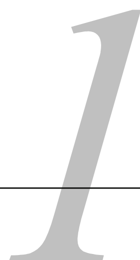
Google App Engine 入门

本部分内容

- 第 1 章　简介
- 第 2 章　入门

第 1 章

简　　介



云计算是一种具有开创性的、令人兴奋的编程及使用电脑的方式。它为软件开发人员创造了巨大的机会：云计算能够为构建新型应用提供一个极棒的新平台。在这一章中，我们将了解一些基本概念：什么是云计算，何时应该使用它，为什么应该使用它，以及应用程序开发人员可以利用哪些类型的基于云的服务。

1.1　什么是云计算

在了解如何使用Google App Engine编写云程序之前，我们从最基础的开始，先弄清楚云计算指的是什么，什么是云，它与桌面计算以及老式的客户/服务器计算模式有什么不同。最重要的是要明白为什么软件开发人员需要关心云，何时需要使用云，以及应该用云来做什么。

1.1.1　云的概念

在现代互联网和万维网的世界中，数据中心分布于世界各地，每个数据中心都拥有成千上万台计算机。使用这些计算机已经成了人们的日常活动，我们通过计算机与他人聊天、发送电子邮件、玩游戏、读博客、写博客，这些活动其实是以浏览器作为客户端，去访问在服务器端运行的程序。

但是，程序实际上在哪里运行呢？数据存放在哪里？服务器在哪里？它们总归位于某个地方，放在某个数据中心，呆在世界的某个角落。用户并不知道在哪里，更重要的是，用户不用去关心，也根本没有理由去关心。用户在意的是在需要的时候要能够访问到这些程序和数据。

让我们看一个简单的例子。几年前，我开始写博客。（该博客虽然已经搬迁走了，但仍然是个很好的例子）。开始时，我使用Google的Blogger服务。每天，我会打开网络浏览器，进入http://goodmath.blogspot.com/admin，然后开始写作。写完后点击“发表”按钮，博客的内容就会呈现给我所有的读者。从我的角度来看，它就是这么工作的。我只需要网络浏览器以及URL地址，就能够写博客。

在后台，Blogger是在Google某数据中心运行的一款复杂软件。它承载了数十万的博客，并且每天都会有数百万的用户来访问这些博客。从这个角度看，显而易见，支撑Blogger的软件运行在很多台计算机上。有多少台呢？我们不知道。实际上，它甚至都可能不是一个固定的数目——

当访问用户不很多时，就不需要在很多机器上运行该软件；当越来越多的人开始用它时，就逐步需要更多的机器了。运行这个软件的机器数目是变化的。但是，从用户的角度来看——不管是博客的作者还是博客的读者，都不需要关心机器的数量这一问题。Blogger是一项服务，并且能够正常工作，这就够了。当我想写博客时，就可以进入Blogger编写；人们只要进入我的博客网页，就可以阅读它。

这就是云的基本理念：程序和数据在某个地方的计算机上，用户不需要知道也不需要关心这台计算机在哪里。

为什么将这些资源的集合称为云？云是大量微小水滴的集合。有些水滴会落在我的院子里，滋养树木和草坪，有些会流入供给我饮用水的水库里。而云本身由各地蒸发的水分形成。我只希望在院子里有足够的水分可以滋养植物，而且水库中蓄水充足让我饮用，我才不关心是哪片云带来了降雨，所有的云对我来讲都是一样的。我不关心水来自于地球的哪个地方，它们都只是水——每个水滴几乎完全相同，我看不出区别。只要有充足的水，我就心满意足了。

所以，试想世界各地的各类数据中心所在的公司都设有大量的计算机，就如同云。很多网络计算方面的最大参与者（包括Google、Amazon、Microsoft、IBM和Yahoo）都有几千台机器接入网络，运行着各类软件。每个这样的数据中心都是一片云，每个处理器、每个硬盘都是云中的水滴。在云的世界中，开发者编写程序时，不知道程序会在哪个计算机上运行。开发者不知道存储数据的硬盘在哪里，而且也不需要关心。开发者只需要清楚自己需要多少"水滴"。

1.1.2　云与开发者

云计算从根本上改变了过去计算机和软件的使用方式。传统上，如果用户想要运行某个应用程序，会去买一台计算机和一些软件，在自己家里安装好软硬件，然后运行程序。用户需要考虑该运行什么操作系统，自己安装好软件，并且需要维护计算机——跟踪软件升级、安全、备份等。

有了云计算，用户不需要再做上述工作。使用云的话，你只需要购买想要的应用程序的访问权，然后就可以在任何地方访问该应用程序。安装软件、维护程序运行所需的软硬件、保持数据安全可靠，这些问题都不需要你来考虑。在云计算环境下，你所购买的软件就是服务。如果需要比普通用户更多的存储空间，你可以从服务提供商处购买额外的存储空间。如果这样意味着需要购买和安装新的硬盘，那也由服务提供商负责。你只是从他们那里购买了存储服务，具体如何提供服务则是他们的问题。你告诉他们你需要什么，既包括实际的物理层面（"我需要1TB存储空间"），也包括略为抽象一点的服务质量层面（"我需要确保存储是事务性的，在我提交一个修改后，数据不会丢失"）。你告诉云计算服务商你有什么需求，他们就会卖给你满足这些需求的服务。

这就意味着，当开发者做云计算软件开发时，不再是购买计算机并在其上运行软件，而是将工作拆分为基本的"积木块"，然后从服务提供商那里购买这些积木块，最后根据自己想要搭建的系统将积木块组合在一起。

这些"积木块"就是开发者运行程序或者执行任务所需的资源，包括如处理时间、网络带宽、磁盘存储空间、内存等内容。作为云计算的使用者，你不需要关心这些资源位于何处。你只需知

道自己需要什么，然后从能够提供最便捷服务的提供商处购买即可。

对于开发者而言，云计算引入了更为巨大的变化。当开发者做云计算开发时，他并不是在实现一个软件，然后销售给客户，而是在为他的客户创建一个可供使用的服务。了解这其中的区别至关重要：开发者时刻要谨记是要提供给用户一种服务，而不是给他们一个要安装在他们的计算机上运行的独立应用程序。客户会根据他们要完成的任务选择服务，所以你的应用程序要根据这个任务来设计，并且必须尽可能以最灵活的方式提供相应的服务。

例如，如果开发者想为桌面计算机创建一个待办事项列表的应用程序，这是一个相当简单的过程。虽然用户界面的布局可能有很多种方式，但是创建这么一个系统的基本思路是显而易见的。开发者会创建一个用户界面——当然不需要创建多个，而且主要为单用户创建。如果为云开发待办事项列表应用程序，那么，开发者就需要有多个用户界面了：最起码，为通过桌面计算机访问这一服务的人提供一个界面，为通过手机上的移动浏览器访问这一服务的人提供一个界面。开发者可能还要提供一个开放式接口，以便其他开发人员能够用它来创建其他设备的客户端。另外，开发者需要针对多用户进行设计，因为应用程序一旦发布到云中，虽然只是单个应用程序，但却会被很多人使用。因此，开发者设计程序时要有这个准备，即使用户不会同时使用该应用程序工作，它也是一个多用户系统。

对于开发者而言，云计算最令人兴奋的一面是它的可扩展性。当开发者在云环境进行开发时，可以只编写一个供一两个人使用的简单程序，然后，甚至无需改变任何一行代码，就可以扩展到支持数百万用户。程序本身是与使用规模无关的，开发者编写的应用程序，被几十个用户使用和被上百万用户使用效果是一样的。随着用户的增加，开发者所需要的只是购买更多的资源，让程序仍然可以正常工作。开发者可以从运行在云中的一台服务器上的简单程序开始，通过增加资源而扩展到支持数百万用户。

1.1.3 云计算与客户/服务器计算

在很多方面，基于云的软件开发的基本方式和客户/服务器计算编程类似。两者都是基于同样的思想：应用程序并不真正运行在用户自己的计算机上。用户的计算机提供了访问应用程序的窗口，但是，并不直接运行应用程序本身，用户在自己的计算机上所需做的全部工作只是运行某种用户界面。真正的程序运行于其他地方被称为服务器的计算机上。之所以要使用服务器，归根结底都是因为用户的本地计算机没有运行该程序所需的资源，而在其他地方更容易获得这些资源，从而能够更便宜、更快速、更方便地运行该程序。

云开发和客户/服务器开发的最大区别在于用户知道的范围不同。在传统的客户端-服务器系统中，用户可能把一台特定的计算机作为服务器，程序就运行在该服务器上。这台服务器可能不在用户面前的办公桌上，但是用户知道它在哪里。例如，我在大学时使用过的第一台大型机是名为“Gold”的VAX 11/780，它位于希尔中心的罗格斯大学计算实验室。在亲眼看到它之前，至少一年的时间里，我几乎每天都在使用它。除Gold外，这个数据中心至少还有三十台计算机：若干台DEC 20，几台Pyramid，一台S/390，还有一群Sun服务器。但是在这些机器中，我只使用Gold。

我编写的每个程序，都专用于在Gold机上运行，而且这也是我能够运行程序的唯一的地方。

而在云中，用户却并不限定在一台特定的服务器上。用户拥有计算资源，但那是有人租给他一定数量的计算资源，这些资源位于某个地方的一些机器上，用户并不知道它们在哪里，也不知道它们是什么类型的计算机。这些机器可能是两个大型机，每个有32个处理器以及64GB的内存，也可能是64个极小的单处理器机器，每个只有2GB的内存。运行这些程序的计算机可能自身带有超大容量的硬盘，也可能是无盘工作站，需要访问专用存储服务器上的存储空间。这些对你这个云用户并不重要。你已经得到了购买的资源，而且这些资源就是你所需要的，它们的位置着实无关紧要。

1.1.4 何时用云开发

综上所述，现在我们知道了什么是云。这是一个思考计算模式的革命性方式：云是一种由服务器所组成的世界，用户可以在其上构建应用程序；云也是一种由服务所组成的世界，用户可以构建这些服务，也可以使用这些服务构建其他的东西。现在的问题是，何时该用云？

用户可以编写几乎任何一个想在云中实现的应用程序。事实上，不少人坚信，一切尽在云中，不再需要为独立的个人计算机再开发什么应用程序 。我还没有那么乐观，诚然，许多应用程序非常适合放在云里，但这并不意味着云这个平台就能完美地容纳一切。用户当然可以在云中构建任何应用程序用作服务，但是，有时这样做要比开发一个独立运行的应用程序困难得多。

在云中构建以下3种应用程序是合乎情理的。

- **协作型应用程序**

  如果用户正在构建的应用程序将被很多一起工作的团队所使用，需要进行数据共享、交流或合作，那么用户确实应该在云中构建此应用程序。协作是云的原生态。

- **服务**

  问一问“这个应用程序是干什么用的？”，如果最自然的答案听起来就像是一种服务，那么就该考虑云应用。应用程序和服务之间的区别是微妙的，我们可以把几乎所有的东西描述为服务。这里的关键问题是，什么是它最自然的描述。如果我们想要描述的是桌面版的iTunes应用程序，我们可以说：“它可以让人们管理自己的音乐收藏。”这听起来确实像是服务。但是，这一描述遗漏了iTunes桌面应用程序的关键属性：它管理用户计算机上的音乐文件，并可以通过串行电缆与iPod上的音乐文件同步。很显然，后面的描述方式意味着它是一个桌面应用程序，而不是云应用程序。

  另一方面，如果你看一下类似于eMusic的应用，你会得出不同的结论。eMusic是一个以订阅为基础的网站，用户可以浏览一个巨大的音乐库，然后每月购买一定数量的歌曲。eMusic显然是一个服务：它允许人们在成千上万的音乐曲目中进行搜索，为他们提供各种功能，如听取音乐片段、阅读各种评论、发表对所听过的音乐的评价、根据用户个人喜好推荐新音乐，并最终选择购买的东西。这显然是一种服务，放在云中合乎情理。

- **大型计算**

  你的应用程序是否需要执行海量的计算，但你却无法承担购买专用计算机的费用？如果

是这样，那么你可以通过云计算来购买服务器“农场”的时间来运行你的应用程序，这种方式经济实惠。对于像遗传学研究人员这些人来说，这真是棒极了，因为他们需要执行海量的计算，却没有资金或其他资源来为自己建立一个专用的数据中心。于是，他们可以购买商业数据中心的机时，与许多其他用户以共享的方式使用这些数据中心。

1.2　云计算编程系统

在云中进行编程的方式有多种。在真正开始编写程序之前，先快速浏览几个例子，简单了解一下几种可行方案。

1. Amazon EC2

Amazon提供了各种基于云的服务。他们的主要编程工具被称为EC2，即弹性计算云（Elastic Computing Cloud）。

事实上，EC2包括一系列相关服务。我们可以拿App Engine做个对比。App Engine提供的是一个单一的、功能极为集中的API套件，而EC2完全不管具体用什么编程API。EC2提供了数百种不同环境：用户可以使用Linux、Solaris或Windows服务器来运行其应用程序；可以使用DB2、Informix、MySQL、SQL Server或Oracle来存储其数据；可以使用Perl、Python、Ruby、Java、C++或C＃来实现其代码；可以使用IBM的WebSphere或sMash、Apache JBoss、Oracle WebLogic或微软的IIS来运行其程序。根据用户喜欢的每种组合，以及计划使用的每种资源（存储空间、CPU、网络带宽）数量，成本不尽相同，既有低廉的CPU小时0.10美元和每GB带宽0.10美元，又有高端的每CPU小时0.74美元。

2. Amazon S3

Amazon还提供了另一种极为有趣的云服务，它与大多数云服务有很大不同。它是一个纯粹的存储系统，名为S3，即*简单存储服务*（Simple Storage Service）。S3既不运行程序，也不提供任何文件系统，更不提供任何索引，它只是一个纯粹的块存储，可以给用户分配一个存储块，存储块拥有一个唯一的标识符，然后用户就可以使用该标识符对此块进行读取和写入。

人们已经创建了使用S3存储方式的各种系统，如基于网络的文件系统、本地操作系统的文件系统、数据库系统以及表存储系统。作为以云资源为基础的范例，S3是一个非常好的例子：存储所涉及的计算与实际的数据存储本身完全分离。用户需要存储空间时，可从S3购买一定的存储空间；需要计算时，可购买EC2的资源。

S3是一个真正令人着迷的系统。它非常专一，只做存储这一件事情，并且采用了一种极其狭窄的方式。但其重要意义在于，这正是云的真谛。S3是一个绝对专一的服务，它只为用户存储数据。

S3的收费基于两个标准：用户存储数据量的大小，用户存储和读取数据所使用的网络带宽。Amazon目前的收费为，每月每GB存储空间0.15美元，带宽资源大约是，上传每GB为0.10美元，下载每GB为0.17美元。

另有相关消息称，Google提供了一个非常相似的云服务，名为“Google开发者存储”，该服务在Google云中复制了S3的基本特性。

3. IBM按需计算

IBM提供了一个云服务平台，该平台基于IBM的网络服务开发套件，采用WebSphere、DB2和Lotus协作工具。这个环境与EC2上基于IBM的环境相同，但它运行于IBM的数据中心，而不是在Amazon的数据中心。

4. Microsoft Azure

Microsoft已经开发和部署了一个名为Azure的云平台。它是一个基于Windows的平台，采用的是标准的网络服务技术（如SOAP、REST、Servlet和ASP）和微软专有API（如Silverlight）的组合。这种组合的结果是，用户可以创建一个异常强大的应用程序，非常像标准的桌面应用程序。但其缺点是，由于该应用程序与Windows平台紧密联系在一起，所以，应用的客户端主要运行在Windows平台上。虽然有其他平台的Silverlight实现，但是，这类应用往往只有在Windows平台上才是最可靠的，只有在Internet Explorer中才是功能齐全的。

因此，这就是云。既然知道了云的概念，下面我们就开始学习如何在云中构建应用程序。Google已经组建了一个很了不起的平台，名为App Engine，可以供用户构建和运行自己的云应用程序。

在本书的其余部分，我们还将详细了解一些用于构建基于云的网络应用程序的关键部件。我们将开始使用Python，用Python来学习基础知识非常棒，它可以让读者看到发生了什么事情，并且很容易快速尝试不同的方法，看看会发生什么。

我们将从基本的构建块（如HTTP、服务和处理程序）开始，贯穿读者用Python构建Google App Engine应用程序所需的各项技术的整个过程。然后，我们将了解如何利用App Engine的数据存储服务，在云中实现数据持久化存储。接下来，我们再了解如何采用HTTP、CSS和AJAX为应用程序构建用户界面。

然后，我们将暂时将内容从Python转移到Java上来。在我看来，用Java构建复杂的应用程序会更加方便。这并不是说Python不能或不应该用于高级的App Engine开发，只是我倾向于使用Java。并且App Engine提供了一个精彩绝伦的架构GWT，它从基于网络的云应用中抽象出了大部分的样板基础工作，使得用户可以把重点放在其感兴趣的部分。我们将花一些时间来学习如何使用GWT创建漂亮的用户界面，以及如何使用GWT的远程过程调用服务来实现AJAX风格的通信。

最后将介绍真实网络开发中最复杂的方面。我们将了解以下有关细节：如何使用App Engine的数据仓库服务来制作复杂系统，如何使用类似cron的机制来实现服务器端的处理和运算，以及如何将安全性和身份认证整合到用户的App Engine应用程序中。

在下一章中，我们就将正式开始学习App Engine，届时将首先介绍如何建立一个App Engine的账户，然后是如何设置用户计算机上的软件，用来构建、测试和部署采用Python编写的App Engine应用程序。

1.3　致谢

写一本书，是一个长期、艰难的过程，没有人能独自做完这件事。完成这本书，花费了很多人的很多时间。

我想要感谢以下人员。

- 技术审稿人Nick Johnson，Scott Davis，Fred Daoud，Lyle Johnson，Krishna Sankar，Dorothea Salo，感谢他们的投入和反馈。
- 本书的编辑Colleen Toporek。回想我无休止拖稿、文思枯竭，以及可怕的拼写错误……感谢她的宽容，从而保证了该书的正常进展。
- Google的App Engine团队，感谢他们构建了如此神奇的系统，为我提供了本书写作的源泉。
- 我的妻子和我绝对顽皮的孩子们，感谢她们在我埋头键盘工作的时间中对我的支持。

第 2 章

入　　门

2

在这一章中，我们将初步了解并开始使用Google App Engine。我们将学习如何完成如下工作：

- 设置Google App Engine账户；
- 下载并安装Google App Engine SDK（软件开发工具包）；
- 创建一个简单的Google App Engine应用程序；
- 在本地测试应用程序；
- 在云中部署和监视Google App Engine应用程序。

这章不是本书中最令人兴奋的章节，但它是开发者能够进一步获得感兴趣内容的必经之路。当然，本章也包括一两个有趣的例子。

2.1　设置 Google App Engine 账户

为了使用Google App Engine编写云应用程序，开发者需要做的第一件事就是开通一个App Engine账户。开发者进行云开发时，需要为应用程序租用计算和存储资源。App Engine 账户为开发者提供了一组基本的免费资源，以及在需要时可以购买更多不同种类资源的机制。

创建一个Google App Engine账户是免费的。一个基本的、免费的App Engine账户可以供用户运行10个应用程序，并且具有如下特点：

- 每天6.5小时的CPU时间；
- 每天10GB的上行带宽和10GB的下行带宽；
- 1GB的数据存储空间；
- 每天可发送2000封电子邮件。

如果开发者有更多需求，可以购买各种额外资源。

要建Google App Engine账户，首先要有一个标准的Google账户。如果开发者使用了Gmail或者iGoogle，就已经有了标准账户了。如果没有，只需要访问，选择屏幕右上角的“注册”，然后点击“现在创建账户”的链接即可。

创建好Google账户之后，开发者就可以在浏览器中打开http://appengine.google.com，开始使用Google App Engine了。开发者将会看到一个标准的Google登录界面，然后用其Google用户名和

密码登录即可。第一次进行该操作时，开发者需要通过手机短信进行身份认证。为了防止垃圾邮件发送者设立App Engine账户，Google设置了使用唯一电话号码注册的机制。别想蒙混过关，一个电话号码只能设置一个App Engine账户——某号码一旦被使用，用户便不能再使用该号码创建其他账户。

CPU 时间计算

开发者每天可获得 6.5 小时的免费 CPU 时间。但是，如果开发者购买了 CPU 时间，随着工作的开展，开发者最终使用的可能远不止于此，甚至最终一天使用的 CPU 时间会超过 24 个小时。在 Google App Engine 中，开发者的应用程序不只在一台服务器上运行，而是运行在 Google 的数据中心。每个发来的请求都会被路由到集群的某台机器上。因为可以有多个用户同时访问开发者的应用系统，所以，应用程序使用了多台物理计算机的 CPU 时间。开发者要为运行其应用程序的所有计算机的 CPU 时间总和付费。这样一来，开发者一天使用的 CPU 时间可能不止 24 小时。

开发者填写完表格后，可以在浏览器中看到一个新页面，要求输入认证码。在10分钟内，开发者将会收到包含认证码的手机短信。输入认证码，就可以使用该账户了。

2.2　设置开发环境

拥有Google App Engine账户后，首先要做的就是创建一个应用程序。而自此开始，云过程的开发和通常的应用程序开发已经稍有不同了。要编写一个在自己计算机上运行的新程序，开发者只需要打开编辑器，然后输入代码即可。而编写云应用程序，开发者需要在云服务器上注册自己的应用程序，以便为其运行创建空间，并获取在该空间工作所需要的工具。

在开发者下载Google App Engine工具之前，要确保自己的计算机上已经安装了Python。Python语言现在处于不稳定状态，不断演变出被显著改写的版本。因此，日常使用中会有多个互不兼容的Python版本。对于App Engine而言，开发者需要使用Python 2.5。所以，必须确保安装的是正确的Python版本。开发者如何在不同的操作系统上安装Python用来开发App Engine服务，不是本节讨论的内容，但是，如果开发者访问Python的主页http://python.org，就可以找到最新的安装指南。

开发者还需要一个文本编辑器或者IDE来编写代码。有很多优秀的免费工具可以使用，开发者只需要挑选适合自己的那一款，并确保将它安装好即可。

当开发者安装好了编写Python程序所需的工具后，就可以登录在上一节创建的App Engine账户，点击“创建应用程序”（Create an Application）按钮，下载Google App Engine Python SDK。然后开发者将看到一个表单，显示的是应用程序的名称和描述。该表单看起来和图2-1中的表单类似。（由于App Engine经常更新，所以确切的表单可能显示出来略有不同。）

图2-1 Create an Application表单

为了创建自己的应用程序，开发者需要向Google App Engine服务提供一些信息。

- **应用程序标识符**

  应用程序标识符（Application Identifier）是开发者的应用程序的唯一名称，以区别于任何其他App Engine用户所运行的任何一个应用程序。该名称将作为访问该应用程序的URL。由于这个属性不能被开发者修改，因此务必谨慎选择！开发者可以输入一个名称并点击“检查是否可用”（Check Availability）按钮进行检查，以确保该名称没有被其他人使用。我建议开发者为其应用程序名称选择一个私有前缀，这样做容易避免与其他人的应用程序发生名字冲突，并且在App Engine程序世界里，可以给开发者编写的一系列应用程序赋予一个通用的标识。为本书创建的所有应用程序，我都使用了`markcc`前缀。对于后文将遇到的示例应用程序，我选用的名称为`markcc-chatroom-one`，因此，该示例应用程序的URL是http://markcc-chatroom-one.appspot.com。

- **应用程序标题**

  这是所有应用程序的用户都会看到的程序名称，并且该名称会在登录页面显示。在示例中，我使用了MarkCC’s Example Chatroom（MarkCC示例聊天室）作为应用程序标题（Application Title）。开发者可以从控制面板随时修改应用程序的标题。

- **安全和身份认证设置**

  开发者可以为应用程序进行初始的安全性和身份认证设置。这个话题在这里就不探讨了，我们将在第17章中再讨论。

- **服务条款**

  开发者必须先接受Google的服务条款（Terms of Service），才能使用Google App Engine创建应用程序。请开发者花点时间通读这些条款，了解自己所做的承诺以及Google给开发人员提供的承诺，然后点击复选框，表明接受这些条款。

填写完该表格，单击Save按钮，Google App Engine就会创建崭新的应用程序架构。开发者保存了初始的应用程序描述后，会看到一个控制面板，该面板用于构建和监视应用程序（图2-2）。

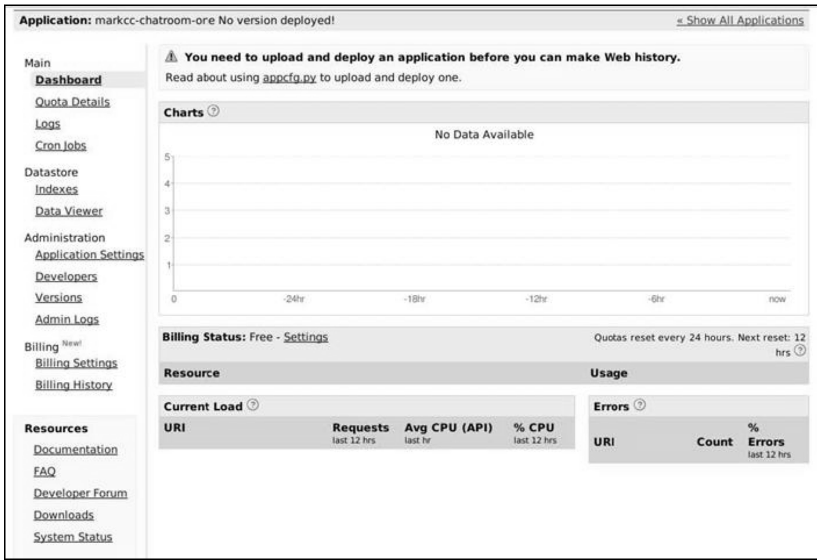


图2-2　Google App Engine控制面板

看到应用程序控制面板，开发者就可以准备开始编程了。不过要注意，云程序的开发和其他的程序开发不同。开发者不能编辑Google App Engine服务器上的文件，需要在本地编写程序，然后使用一个管理脚本将本地程序传送到App Engine环境中去运行。

下一步就是获取工具。在控制面板的左下角，开发者会看到一个标记为“资源”的区域。这个区域包含了开发者学习和使用Google App Engine时需要的软件、论坛、文档的链接。现在，点击“下载”链接，为Python下载合适的App Engine SDK版本。下载完成后，紧接着进行安装。根据自己使用操作系统，安装过程会略有不同：对于Windows或Macintosh而言，下载的内容已包含一个自动安装程序，只需运行它，就能完成安装工作；如果开发者使用的是Linux，下载的内容是一个zip压缩文件，需要在恰当的地方解压该文件。

如果使用的是Windows或者MacOS，就可以准备开始使用了；如果是Linux，则需要将解压SDK压缩文件的目录添加到相关路径中。

在SDK中，开发者会使用到以下两个主要程序。

- `dev_appserver.py`：该程序运行了一个模拟的Google App Engine环境，开发者可以使用该环境在本地计算机测试其应用程序。
- `appcfg.py`：该程序用于在使用Google App Engine的云中上传和配置开发者的应用程序。

2.3　开始 App Engine 中的 Python 编程

现在，我们可以开始编程了！

Python App Engine的核心非常简单。主引擎是一个轻量级、安全的CGI执行程序。CGI是执行程序响应HTTP请求的最古老的接口之一。Google App Engine本质上就是纯粹的CGI，其最大的优势在于，开发者曾用Python编写过的任何CGI脚本，都可以应用于Google App Engine。任何为CGI脚本编写的框架和Python库都能用于Google Apple Engine——开发者上传应用程序代码时只需包含所用的框架/库文件即可。

为什么从 Python 起步？

稍后还会就该问题给出更多解释，但我并不是一个超级 Python 迷。我们从 Python 开始学习有若干原因。

首先，Python 是一门非常友好的语言，开发者使用少量的代码就可以完成大量的工作。我们只用几行代码就可以编写出 App Engine 应用程序。Python 需要的基础设施非常少。当开发者想学习如何进行云开发时，Python 就是个极好的开始方式。

其次，我不应该根据自己的爱好来规定开发者应该如何构建自己的应用程序。Python 是一门非常强大、灵活的语言，并且很好地得到了 App Engine 的支持。如果你是 Python 爱好者，那么读完这本书，你应该能够用 Python 构建自己的 App Engine 应用程序了。

再次，我们将要学习一些工具，如 GWT，它们生成了大量代码，为我们的云应用程序处理客户端-服务器的底层交互机制。对于复杂的应用程序开发，这样做可以为我们节省大量的时间。但是，了解幕后原理很重要。

Python 为我们提供了一种探索云应用程序原始构造的良好方法。我们能够看到每个技术细节，将其构建起来，并了解它是如何工作的。当讲到 GWT 时，我们就更容易理解云应用程序是怎么构造的了。

如果开发者喜欢用 Python 进行云编程，那么你在本书中将会学到足够多的知识，能用其进行云编程。但是，即使开发者不使用 Python 编写云应用程序，不管准备使用哪种语言，花些时间通过 Python 探索云应用程序的基本技术也会有助于开发者理解和调试真实的应用程序。

使用Google App Engine工作最简单的方法是应用它自带的框架，即webapp。webapp是一个非常精致、强大的框架，使用起来十分简单、友好。

此外，Google App Engine给开发者提供了对Google服务的访问权限，如登录、数据存储、安全、身份认证，以及应用程序内的付费功能。在本书中，我们将重点放在webapp框架的使用上——但是一旦开发者知道如何使用App Engine工作并执行程序，就可以使用这些服务以及其他框架编写App Engine应用程序了。

本书的大部分内容将围绕一个聊天室应用展开。但在开发聊天室应用之前，我们先开发个云版本的"Hello, World."。这是个运行在云服务器端的简单程序，会生成一个显示在用户浏览器中的欢迎页面。

因为云应用程序通常使用网络浏览器作为其用户界面，所以我们要开发的应用程序必须生成HTML格式，而不仅仅是纯文本。无论何时生成输出，首先需要包含一个MIME头，MIME头只有一行，用于指定后续内容的格式。对HTML而言，后续内容的类型为`text/html`。

Google App Engine SDK希望开发者所有的应用程序文件存储在一个目录层次结构中。我们为第一个程序创建一个名为`chatone`的目录。在该目录中，我们将在名为`chat.py`的文件中编写这个微型欢迎程序：

chatone/chat.py

```
import datetime

print 'Content-Type: text/html'
print ''
print '<html>'
print '<head>'
print '<title>Welcome to MarkCC\'s chat service</title>'
print '</head>'
print '<body>'
print '<h1>Welcome to MarkCC\'s chat service</h1>'
print ''
print 'Current time is: %s' % (datetime.datetime.now())
print '</body>'
print '</html>'
```

为了能够运行该程序，我们需要告诉Google App Engine它是用什么语言编写的，它需要什么资源，代码在哪里，以及如何用代码实现向服务器发送请求。

在Google App Engine中，我们通过编写`app.yaml`文件完成该工作：

chatone/app.yaml

```
application: markcc-chatroom-one
version: 1
runtime: python
api_version: 1

handlers:
- url: /.*
  script: chat.py
```

文件app.yaml总是以一个文件头开始，该头部的字段描述如下所述。

- application

  我们正在构建的应用程序的名称。该名称必须与开发者创建应用程序的控制面板时给出的名称完全匹配。

- version

  一个字符串，用来声明我们的应用程序的版本。该字段主要是给我们提供信息，以便可以进行一些操作，如查询服务器以得到正在运行的应用程序的版本，或者识别引起某些错误的代码版本。开发者可以为该字段配置任意的字符串标识。

- runtime

  我们编写该程序所使用的语言：Java或Python。

文件头之后，我们需要编写一个处理程序子句列表。当Google App Engine服务器收到一个发来的HTTP请求时，通过该列表的描述，决定其应该执行什么操作。服务器会把HTTP请求路由到我们所编写的脚本上。App.yaml文件中的处理程序子句就是用来通知服务器哪些请求会路由到哪些Python脚本的。在每个处理程序子句中，url模式使用正则表达式声明，紧接着是描述收到匹配该模式的请求时应该执行的操作的子句。在我们的例子中，只有一个处理程序。我们的应用程序接收到任何请求都会通过运行欢迎脚本来进行响应，所以url的模式是/.*，表示匹配任意请求。无论何时收到一个请求，我们都希望运行欢迎脚本。所以，代码的操作为script: chat.py，表示"运行名为chat.py的脚本"。

为了测试这个应用程序，首先使用dev_appserver.py在本地运行该程序：

```
$ ls
Chatone
$ ls chatone
app.yaml  chat.py
$ dev_appserver.py chatone
INFO    2009-06-18 23:13:31,872 appengine_rpc.py:157]Server:appengine.google.com
INFO    2009-06-18 23:13:31,880 appcfg.py:320] Checking for updates to the SDK.
INFO    2009-06-18 23:13:31,994 appcfg.py:334] The SDK is up to date.
WARNING 2009-06-18 23:13:31,994 datastore_file_stub.py:404] Could not read \
             datastore data from /tmp/dev_appserver.datastore
WARNING 2009-06-18 23:13:31,994 datastore_file_stub.py:404] Could not read \
             datastore data from /tmp/dev_appserver.datastore.history
INFO    2009-06-18 23:13:32,058 dev_appserver_main.py:463] Running application\
             markcc-chat-one on port 8080: http://localhost:8080
```

随着应用程序的本地运行，我们可以使用网络浏览器进行测试。查看运行dev_appserver.py时输出的最后一行——该行提供了本次会话的URL——本示例中是http://localhost:8080。如果我们在浏览器打开该URL，可以看到类似的内容：

Welcome to MarkCC's chat service

Current time is: 2009-06-18 22:30:47.345731

既然我们知道该程序可以运行，现在就可以运行appcfg.py命令将其部署到云服务器中。appcfg.py是开发人员访问Google App Engine的主要接口，所以它支持很多不同的应用程序。要

发送代码到服务器，可使用appcfg.py的update命令。

```
$ appcfg.py update chatone
Scanning files on local disk.
Initiating update.
Cloning 1 application file.
Uploading 1 files.
Deploying new version.
Checking if new version is ready to serve.
Will check again in 1 seconds.
Checking if new version is ready to serve.
Closing update: new version is ready to start serving.
```

现在，代码就部署在服务器上了。开发者可以通过http://your-app-name.appspot.com网址访问该程序。

即使在这第一个例子中，也可以开始看出Google App Engine编程的基本风格。我们没有使用任何webapp框架的内容，但仍然体现了其基本概念：app.yaml文件指出了收到的请求是如何被路由到组成我们程序的脚本中的，并指出我们的应用程序和用户通信的方法是在用户浏览器中生成HTML内容。

上述琐碎方法的问题在于，生成MIME头以及HTML页面结构这些事情都是通过手工完成的。手工完成这些工作非常繁琐，而且极容易出错。当开发者开始编写交互式应用，解析收到的请求输入时，这种方式的弊端就更加严重。webapp框架提供了处理基本HTTP请求/响应循环的基础架构，能解析收到的请求，生成必要的文件头，管理与网络服务器的通信以发送响应。另外，webapp提供了一组模板处理器供你使用，使得开发者可以创建响应的架构，而无须输出整个HTML结构。只是现在，我们的HTML足够简单，不需要使用模板功能，但在第6章中，我们会详细了解模板的功能。

下面是使用基本webapp架构的欢迎页面应用程序的版本。

chatone/chatonewa.py

```
❶ from google.appengine.ext import webapp
  from google.appengine.ext.webapp.util import run_wsgi_app
  import datetime

❷ class WelcomePage(webapp.RequestHandler):
❸     def get(self):
❹         self.response.headers["Content-Type"] = "text/html"
❺         self.response.out.write(
            """<html>
                 <head>
                   <title>Welcome to MarkCC's chat service</title>
                 </head>
                 <body>
                   <h1>Welcome to MarkCC's chat service</h1>
                   <p> The current time is: %s</p>
                 </body>
               </html>
            """ % (datetime.datetime.now()))
```

```
❻ chatapp = webapp.WSGIApplication([('/', WelcomePage)])

❼ def main():
      run_wsgi_app(chatapp)

  if __name__ =="__main__":
      main()
```


我们快速浏览一下，看看webapp各部分代表什么意思。

❶ 首先，我们需要引入将要使用的webapp框架的各个部分。对这个程序来讲，我们使用了两个基本的webapp构建块：webapp模块本身和一个名为`run_wsgi_app`的webapp函数。

❷ 接下来，我们创建一个webapp的`RequestHandler`。webapp理解HTTP是如何工作的，并提供了用于使用HTTP协议的所有基本元素的实用工具类。云应用程序中的基本操作就是响应用户发出的请求。`RequestHandler`类就是为处理这些请求而创建的——这是开发者使用最多的webapp类。

❸ 在欢迎应用程序中，我们唯一需要处理的HTTP请求是`GET`。在webapp中，开发者提供`RequestHandler`子类的`get()`函数的实现就可以处理该请求。

❹ webapp并不手工生成MIME头等内容，而是给开发者提供一个包含Python映射的响应结构。开发者要想给任何HTTP头设置值，在映射中赋值即可。我们用到的唯一的头是Content-Type，所以将它写入映射中。

❺ webapp并不将生成的结果直接输出到标准输出，而是给开发者提供了一个可管理的缓存输出通道。开发者将响应的内容写到缓存输出通道。App Engine开发者新手最常犯的错误是：他们用以往的旧式输出语句输出HTML，然后质疑为什么看不到输出结果。开发者需要确保自己始终在使用webapp的缓存输出通道。

❻ 为了使用webapp的`RequestHandler`，我们需要创建一个应用程序对象。应用程序对象与`app.yaml`文件非常相似：`app.yaml`文件描述的是如何将收到的请求映射到特定的应用程序脚本上，而应用程序对象描述的是如何获取映射到该脚本的请求，并将这些请求映射到特定的`RequestHandler`类上。我们将对根URL的请求，即对http://markcc-chatroom-one/的请求，映射到欢迎页面请求处理程序上。

❼ 文件的其余部分使用了一个常见的Google用法。为了实际运行应用程序，该脚本需要调用。但是，我们并不是直接执行该语句，而是编写一个主函数，使用间接方式调用它。这个用法使得脚本的复用变得容易。如果我们直接运行`run_wsgi_app`，那么当需要在其他Python代码中引入该脚本时，就会执行该行。增加了条件调用的主函数保证了该脚本只有在专门调用时才会被执行，而不会被引入它的模块执行。

我们可以使用与上一个程序相同的方式部署这一新版本。我修改了Python脚本的文件名，所以还需要更新`app.yaml`文件，以引用新的Python脚本文件。一旦完成这些操作，只需运行`appcfg.py update chatone`就可以完成部署。该命令会上传更新。更新一旦执行完成，就会访问应用程序的URL，随即运行新版本代码。在Google App Engine中，它仍然会显示为版本1，因为我并没有修改`app.yaml`中的版本标识符。

这样粗略的讲解，只是告诉开发者什么是Google App Engine程序开发。在后续的章节中，我们将更加详细地讲解如何使用App Engine开发有趣的网络应用程序。

2.4　监视应用程序

作为应用程序开发人员，你会想去监视该应用程序，以便能够监测有多少用户、耗费多少资源、点击数有多少、哪些脚本使用最活跃、什么脚本在工作、什么脚本没有工作等等。开发者可以使用Google App Engine控制面板实现这些。

Google App Engine控制面板的监视功能对开发者确实很重要。由于云应用程序运行在不受开发者控制的计算机上，开发者不可能只运行分析器或者调试器就能分析出为什么应用程序没有按照预期的方式工作，所以当开发者发现有些部分运行的时间比预期的时间长很多时（只要编写的是正常的应用，这样的事情总会发生），监视器是开发者的信息来源。

例如，在日常工作中，我会为Google构建基于云的数据分析应用程序。 我不止一次地遇到这样的情况，通常运行1个小时的分析程序居然耗费了8个小时，情况明显非常糟糕。通过控制面板我就能知道发生了什么，我查看控制面板提供的信息，发现系统处理的某些数据碎片非常大，远远超出正常水平。这使我顺藤摸瓜找到了问题所在：有人修改了分析器输入数据的代码，而其生成的数据的形式影响了我代码的性能。如果没有工具让我查看服务器集群上发生了什么的话，想找出这个原因将异常困难。

开发者可以监视应用程序的方方面面——从程序运行过程中代码记录的信息，到Google App Engine服务器记录的信息，再到应用程序消耗的资源的详细信息，都在控制面板中。

应用程序控制面板的主视图称为仪表板。仪表板提供了开发者想知道的有关应用程序的状态和资源使用情况的一切内容的摘要，并包含可以得到更多详细信息的链接。仪表板分为四个部分，如图2-3所示。

- **应用程序性能的可视化视图**，这部分可以用来显示应用程序随着时间的变化使用资源的方式。
- **账单状态视图**，显示开发者对每类资源已经使用了多少，以及在账单计划内还有多少可用。
- **负载视图**，显示`app.yaml`文件中声明的不同的URI模式，以及响应每类请求分别耗费了多少CPU时间。
- **错误视图**，显示应用程序中出现的全部错误的基本摘要信息。

最后，控制面板提供了一组有用的链接，列在“资源”（Resources）下边。这些资源包括：开发者论坛，在此开发者可以与Google App Engine的其他开发者以及Google的App Engine团队讨论问题；官方最新的App Engine文档；常见问题的解答。我强烈建议开发者使用这些链接，特别是开发者论坛。开发者开发App Engine程序遇到的大多数问题，与其他App Engine开发人员遇到的问题类似，论坛里会有你需要的答案。

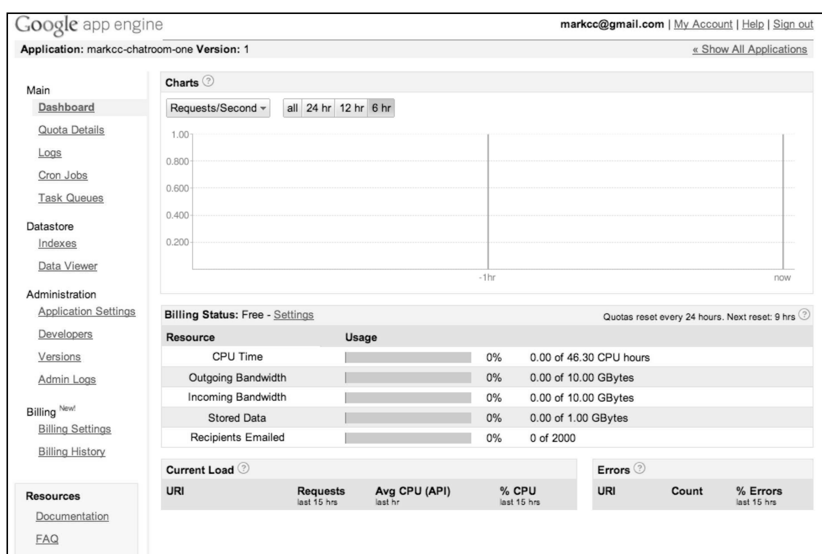


图2-3 控制面板的各部分

除了这些基本的资源信息，在控制面板主视图的左边有一组管理链接，开发者可以进一步详细查看系统的各个方面。

该详细链接提供有用数据的一个例子就是应用程序的日志信息。Google App Engine服务器收到每个请求都会生成一个描述该请求的数据记录。此外，错误也记录在日志里。当开发者的应用程序抛出异常时，异常不会显示给用户，唯一可以看到异常的地方是在日志里。另外，开发者也可以在程序中添加日志记录语句。下面，我们就来看看请求日志。

打开应用程序的控制面板。屏幕左侧包含了各种视图的链接，以及用于监视、控制和管理应用程序的工具集合。最上面的部分是“主标签”，第三个链接是“日志”。点击“日志”链接，切换到显示日志信息的视图。到这一步，如果开发者没有犯任何错误，视图的内容基本上为空——日志的默认视图是应用程序遇到过的所有错误的简洁视图。

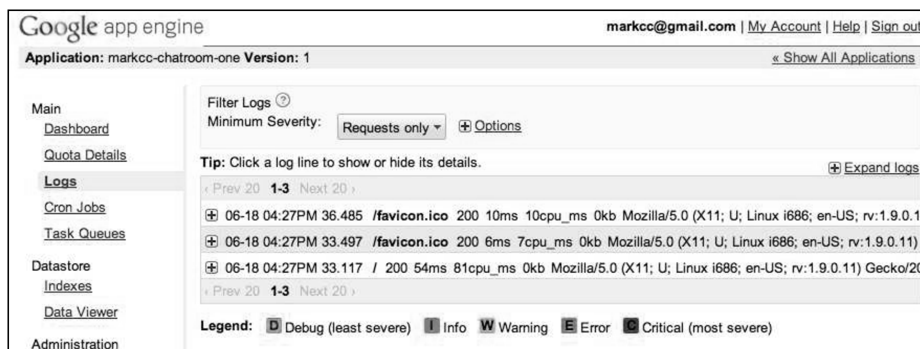


图2-4 请求日志视图

要查看应用程序已经处理的请求，需要使用在日志列表上方标有“最小严重性”的下拉框菜单。打开该菜单，并选择“仅请求”，结果如图2-4所示，显示的是应用程序收到的每一个请求的日志表项列表。

至此就完成了所有设置。开发者拥有了自己的Google App Engine账户，并安装了App Engine工具。已经可以建立App Engine应用程序了。在下一章中，我们将使用刚刚安装的App Engine工具构建一个真正运行在云中的聊天应用。

2.5 参考文献和资源

- Google App Engine开发者指南，http://code.google.com/appengine/docs/。
  针对Python和Java API的官方Google App Engine文档。
- 通用网关接口（CGI），http://www.w3.org/CGI/。
  CGI的官方标准和文档。

Part 2

第二部分

用 Python 进行 Google App Engine 编程

本部分内容

第 3 章

第一个真正的云应用程序

在这一章中，我们将建立第一个有实际意义的云应用程序：一个基本的Python聊天室。通过这个过程，我们将了解如下内容。

- 云应用程序使用的HTTP协议，以及云应用程序如何通信。
- 如何将一个用Python编写的普通非云程序用HTTP框架封装，使其可以在云中工作。
- 在云应用程序中，如何管理数据和变量。

3.1 基本的聊天应用程序

作为一个运行实例，我们将用Python构建一个聊天服务。这是一个很好的例子，因为大家都熟悉它——我们都使用过聊天服务。但即使是这一古老常见的应用程序，它也有许多云服务的典型特点。

云应用与众不同、引人入胜的原因在于：它本质上是多用户的。开发者构建云应用程序，必须要考虑如何应对多用户，以及如何处理多用户数据。

聊天应用程序虽然简单，但却非常典型。要构建一个聊天程序，我们需要考虑多用户之间的交互，需要存储和读取永久型数据，并且针对不同的讨论话题会有多个数据流。考虑好这些，就可以很容易地创建一个简单的版本，然后逐步增加新的功能，展示越来越多的App Engine功能。

在这一部分，我们将忽略用户界面，将注意力集中在后台（我们将在第7章讲述用户界面相关内容）。而现在，我们只编写一个传统的本地聊天应用程序的基本后台，通过函数调用将其与用户界面挂接。在本章我们不实现整个聊天应用程序，但本书的主要内容就是逐步完成聊天应用软件的所有工作。

一个基本的聊天应用程序并不是很复杂。可以想象一下那种典型的聊天程序。聊天程序的用户界面非常简单，它应该有两个窗口，一个可以看到聊天内容，一个用于输入新的文本。聊天内容应包含聊天过程中发送的一系列消息，每条消息用发送者的名称和消息发送的时间作为标识。那么，基本的聊天界面看起来应该与图3-1的模型类似。

现在我们大概知道聊天程序应该是什么样子，可以开始考虑如何去构建它了。在开始剖析如何构建一个云应用程序前，我们首先考虑如何实现一个聊天应用程序的后台作为标准的服务器程序。因此，我们将致力于开发应用程序架构，该架构不使用任何App Engine代码，而是使用标准Python完成聊天程序所需的所有功能。

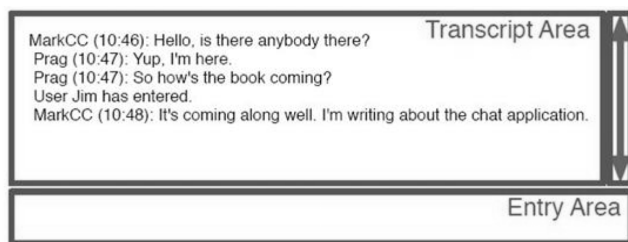


图3-1　聊天界面的模型

我们需要实现什么呢？从基本的聊天内容我们可以看到，聊天系统有一个虚拟空间，用户可以进入和离开该虚拟空间。用户进入后，可以发送消息。任何已发送的消息对所有已经进入该空间的人都是可见的。因此，我们要处理三个基本对象：空间（我们称为聊天室）、用户和消息。

我们希望该空间里能够进行多个会话，从而可使用户决定他们想要谈论的内容，以及谈话对象。我们称能够发生一个会话的空间为聊天室。聊天室里可能发生3件相关的事件：用户可以进入、可以离开、可以发送消息。此外，为了简化，我们并不在每条消息发送出去后就自动更新每个人的聊天内容，而仅仅提供给用户定期查看聊天内容的方式。下面是一个简单的聊天室的实现。该实现不是按照云开发的方式编写的。云应用程序的行为与此完全不同，所以云编程需要有不同的实现。在本书的后续部分，我们会构建一个App Engine的云应用程序，它会使用云中的方式完成本程序的所有工作。到时，读者可以看到它们之间有多大的差别。

basechat.py

```
class ChatRoom(object):
    """A chatroom"""

    rooms = {}

    def __init__ (self, name):
        self.name = name
        self.users = []
        self.messages = []
        ChatRoom.rooms[name] = self

    def addSubscriber(self, subscriber):
        self.users.append(subscriber)
        subscriber.sendMessage(self.name, "User %s has entered." % subscriber.username)

    def removeSubscriber(self, subscriber):
        if subscriber in self.users:
            subscriber.sendMessage(self.name, "User %s is leaving." % subscriber.username)
            self.users.remove(subscriber)

    def addMessage(self, msg):
        self.messages.append(msg)
```

```
    def printMessages(self, out):
        print >>out, "Chat Transcript for: %s" % self.name
        for i in self.messages:
            print >>out, i
```

聊天室需要的下一个元素是用户。用户有一个名称，并注册了一些聊天室。用户可以进入聊天室、离开聊天室，或发送消息。如果他们还没有进入某个特定的聊天室，则不允许向那个聊天室发送消息。

basechat.py

```
class ChatUser(object):
    """A user participating in chats"""
    Def __init__ (self, username):
        self.username = username
        self.rooms = {}

    def subscribe(self, roomname):
        if roomname in ChatRoom.rooms:
            room = ChatRoom.rooms[roomname]
            self.rooms[roomname] = room
            room.addSubscriber(self)
        else:
            raise ChatError("No such room %s" % roomname)

    def sendMessage(self, roomname, text):
        if roomname in self.rooms:
            room = self.rooms[roomname]
            cm = ChatMessage(self, text)
            room.addMessage(cm)
        else:
            raise ChatError("User %s not subscribed to chat %s" %
                            (self.username, roomname))

    def displayChat(self, roomname, out):
        if roomname in self.rooms:
            room = self.rooms[roomname]
            room.printMessages(out)
        else:
            raise ChatError("User %s not subscribed to chat %s" %
                            (self.username, roomname))
```

聊天消息是最简单的部分，它只是来自用户的消息。消息需要包括发送消息的用户、用户想要发送的文本，以及消息发送的时间。

basechat.py

```
class ChatMessage(object):
    """A single message sent by a user to a chatroom."""
    def __init__ (self, user, text):
        self.sender = user
        self.msg = text
```

```
        self.time = datetime.datetime.now()
    def __str__ (self):
        return "From: %s at %s: %s" % (self.sender.username, self.time, self.msg)
```

为了测试聊天消息对象，我们仅编写一个简捷的主程序——也就是说，我们编写的代码具有聊天应用程序所需的功能。在该示例中，我们只是演示程序实际运行时可能是什么样子，而不是进行真正的交互式聊天。我们用该程序创建一些用户，让用户订阅一些聊天室，并发送消息。

basechat.py

```
def main():
    room = ChatRoom("Main")
    markcc = ChatUser("MarkCC")
    markcc.subscribe("Main")
    prag = ChatUser("Prag")
    prag.subscribe("Main")

    markcc.sendMessage("Main", "Hello! Is there anybody out there?")
    prag.sendMessage("Main", "Yes, I'm here.")
    markcc.displayChat("Main", sys.stdout)

if __name__== "__main__":
    main()
```

当我们运行该程序时，得到如下信息：

```
Chat Transcript for: Main
From: MarkCC at 2009-06-27 15:10:51.181035: User MarkCC has entered.
From: Prag at 2009-06-27 15:10:51.181194: User Prag has entered.
From: MarkCC at 2009-06-27 15:10:51.181218: Hello! Is there anybody out there?
From: Prag at 2009-06-27 15:10:51.181237: Yes, I'm here.
```

它远不够完美。时间标签过于详细，而且文本没有格式化，不易阅读，但是基本的聊天功能（聊天室、订阅、用户和消息）都有了。

3.2 HTTP 基础

刚才设计并实现聊天室的架构，对传统应用程序来说是一种合理的开发方式。但是若要设计一个在云中运行的应用程序，则还需要另一个步骤。在传统的应用程序中，开发者要思考如何设计应用程序的后台数据处理，而且必须设计用户界面。编写云应用程序时，仍然需要做这些事情，同时，还需要设计应用程序的协议。

云应用的后台运行在数据中心某处的一台服务器或一组服务器上，而用户界面运行在用户的网络浏览器中。协议的工作就是描述后台和前台如何通信，从而生成一个看起来好像是在用户的浏览器里运行的应用程序。

大多数云应用程序，以及几乎所有的App Engine应用程序，都是基于一种称为HTTP（超文本传输协议）的基本协议构建的。对云应用程序而言，开发者需要设计一种置于HTTP上层的协议。这里的分层只是意味着：该协议构建的目的是使应用程序协议中的每个交互都是用HTTP交

互进行描述的。这是云应用程序开发与传统的应用程序开发不同的关键因素之一，云应用程序紧密围绕客户端和服务器的HTTP交互建立，将应用程序正确地叠加在HTTP之上是构建吸引人的云应用的关键，这样才能够提供良好的用户体验。如果开发者不习惯HTTP的话，可能就会觉得它有点麻烦，但稍后在本书中，读者将会看到如何使用HTTP交互来构建几乎任意类型的应用。

也许读者已经熟悉HTTP了，但仍值得花时间来回顾一下，因为了解HTTP的基本知识对理解App Engine应用程序的工作原理非常重要。下面，在学习如何为聊天室组建应用程序协议前，我们先回顾一下HTTP的基本知识。

HTTP是一种简单的请求/响应式的客户端-服务器端协议。换句话说，它是一个支持双方通信的协议。其中一方称为*客户端*，另一方称为*服务器端*。客户端和服务器端有不同的行为。在HTTP中，客户端向服务器端发送请求，是真正的通信发起方；服务器端处理来自客户端的请求，并发送响应。HTTP做的几乎所有工作就是，描述客户端向服务器端发送请求以及得到响应的方式。

为了使事情更加简化，每一个HTTP请求都汇聚在一个*资源*上。资源是网络中赋予了名称的任意事物。资源还有一个名称是统一资源定位符，即URL。URL是一种文件名，不同的是它可以被用来命名很多不同的东西：文件、程序、人、过程或者你能够想象到的几乎任何事物。每一个HTTP请求或者要求从资源中回收数据，或者向资源发送数据。

每个来自客户端的HTTP请求调用服务器上的方法。（不要被这些术语迷惑：虽然称之为“方法”，但HTTP实际上并没有面向对象的内容。）HTTP有以下四个基本方法（另外还有大约数十个扩展，因为网络应用程序不需要这些扩展，所以我们先不描述它们）。

- `GET`　要求服务器从资源获取一些信息，并且将其发送给客户端。
- `HEAD`　要求服务器告诉客户端有关资源的信息。基本上和`GET`请求类似，不同的是应答只包含元数据。开发者可以使用该请求完成如查询“资源有多大？”等类似工作，而不需要下载整个资源。通常，大多数网络应用程序不使用`HEAD`方法，但是它有时非常有用。
- `PUT`　把数据存储到资源里。客户端将信息发送到服务器端，将信息存储到目标资源里，服务器端应答只需要说明数据是否正确存储即可。
- `POST`　将数据发送给服务器端上的程序。`POST`请求有点奇怪。`PUT`和`POST`并没有太大不同。两者的区别要追溯到万维网早期，即人们在自己独立的计算机上运行网络服务器的时期。在早期网络服务器的实现中，所有的`GET`和`PUT`请求解析为取回和存放文件。所以为了能够在网络服务器上运行程序，开发者需要一个不同的HTTP协议请求，该请求专用于要求开发者运行程序。在现代系统中，`PUT`和`POST`几乎是通用的。

开发者使用网络浏览器生成的每个HTML请求，以及App Engine服务要处理的每个HTML请求有以下3个部分。

- **请求行**：包括该请求的HTTP方法，后接资源的URL地址，最后是协议版本声明。浏览器生成大多数请求使用的方法是`GET`，并且版本声明通常是`HTTP/1.1`。
- **一系列头部行**：由有关请求的元数据组成（如我们在2.3节“开始App Engine中的Python编程”中，欢迎应用程序使用的`Content-Type`声明）。来自浏览器的多数请求都会有头部行，告诉服务器，用户使用什么浏览器（`User-Agent`头部）以及一些用户标识符（`From:`

头部）。头部也可以包含cookie索引、语言标识符、网址等。实际上，任何内容都可以放在头部，因为服务器会忽略其不识别的头部。

- **主体**：包含任意的数据流。

头部的结束和主体的开始通过空行分隔。通常，GET请求和HEAD请求的消息主体是空的。例如，下面是一个简单的GET请求：

```
GET /rooms/chatter HTTP/1.1
User-Agent: Mozilla/5.001 (windows; U; NT4.0; en-US; rv:1.0) Gecko/25250101
Host: markcc-chatroom-one.appspot.com
```

当开发者给服务器发送HTML请求时，服务器应答一个类似结构的消息。不同之处在于在请求行的地方，服务器以状态行开始响应消息。状态行的开头是状态码和状态消息，告诉开发者请求是否已被正确处理，如果没有，是什么地方出错了。开发者的云服务需要处理的典型状态行与HTTP/1.1 200 OK类似，其中HTTP/1.1告诉开发者服务器使用的协议版本，200是响应的状态码，OK是状态文本。

3

状态码总是由3个数字组成。第一个数字是常见的响应类型。

1 表示“信息响应”。

2 表示成功。

3 表示重定向，告诉客户端到别的位置去查找资源。（实际上，重定向是发给客户端的消息，表示“如果你想要该资源，在这个URL上查找它”。）

4 表示客户端错误（例如，404的意思是客户端请求的资源在该服务器上并不存在）。

5 表示服务器错误（例如，该请求引发服务器执行一个脚本，而这个脚本崩溃了）。

例如，下面是一个对上述GET请求的成功应答：

```
HTTP/1.1 200 OK
Date: Sat, 26 Jun 2009 21:41:13 GMT
Content-Type: text/html Content-Length: 123

<html>
  <body>
    <p> MarkCC: Hello, is there anybody out there?</p>
    <p> Prag: Yes, I'm here.</p>
  </body>
</html>
```

让我们试着贯穿一个完整的请求/响应循环。假设我们的应用程序使用POST提交聊天消息。请求可能看起来如下所示：

```
POST /submit HTTP/1.1
User-Agent: Mozilla/5.001 (windows; U; NT4.0; en-US; rv:1.0) Gecko/25250101
Host: markcc-chatroom-one.appspot.com
From: markcc@phouka.local

<ChatMessage>
  <User>MarkCC</User>
  <Date>June 26, 2009 16:33:12 EDT</Date>
```

```
  <Body>Hello, is there anybody out there?</Body>
</ChatMessage>
```

HTTP 状态码

HTTP标准具有大量的服务器结果状态码。以下这些是开发者最常遇到的状态码。

- **200成功**　请求已被成功完成。消息的主体将包含成功请求的结果。
- **301永久移动**　请求已被永久移除，并且该请求以及对该资源以后的所有请求都应该发送到新的地址。
- **303查看其他**　对于该请求，可以通过对其他URL进行GET操作得到结果。URL可以在消息头部的位置找到。这通常用作PUT请求的结果，当PUT处理成功后，服务器告诉客户端在哪里查看操作结果。
- **401未经授权**　请求是有效的，但是用户没有提供任何身份认证数据，表明自己应该被允许看到该资源。用户可以使用某些其他请求，先获得一个认证码，然后重试该请求。
- **403禁止**　请求是有效的，但是用户不被允许访问指定的资源。这与401相类似，但表示用户已经通过身份认证，仍然无法访问该资源，甚至不被允许访问此资源。
- **404未找到**　在指定的位置没有找到该资源。
- **500内部服务器错误**　服务器在处理请求过程中发生的任何错误最终都将产生一个500错误。对于开发者实现的App Engine服务，当开发者的请求处理程序崩溃或产生异常时，客户端浏览器会收到一个500状态码。
- **501未实现**　该请求希望执行某个操作，但是服务器不支持该操作。通常，如果开发者在POST请求中拼错URL地址，则会出现这种错误。

如果消息发送成功，应答可能类似如下内容：

```
HTTP/1.1 303 See other
Date: Sat, 26 Jun 2009 21:41:13 GMT
Location: http://markcc-chatroom-one.appspot.com/
```

3.3　聊天应用程序到 HTTP 的映射

为了使我们的Python聊天应用程序作为App Engine的网络应用程序工作，需要将该应用程序的基本操作映射到HTTP的请求和响应上。

在这个版本中，我们先不处理订阅：因为只有一个聊天室，所以如果用户连接到该聊天应用程序，他就已经在聊天室里了。而且现在我们也不必考虑用户的进入和离开。

假如你在使用聊天室，那么你希望能够做哪些事情呢？

首先，你要在聊天室里看到所有的新消息。拿HTTP术语来说就是：聊天室是一个资源，你想看到它的内容。这自然适合使用GET：你希望浏览器能够取回聊天室的内容，然后显示给你。

你还希望能够发送消息，所以这就需要浏览器能够作为一个活动的实体与聊天室交谈，并告诉聊天室应用程序你有话要说。再者，聊天室是资源，但是这次你想要和它交谈。这是一个PUT或POST操作。在决定要使用PUT还是POST时，可以问问自己，你究竟是想要替换资源的内容，还是想与资源进行交互？向聊天室发布一条消息显然属于后者，我们不想替换聊天室的内容，而是要与它交谈，并且告诉它有一条新消息要添加到会话中。因此，发送一个新消息绝对是一个POST操作。

这样，我们得到了应用程序所需的框架。我们要有一个资源，即一个聊天室。用户可以GET该资源，查看当前的聊天状态。我们还需要另外一种资源，它是个活动的进程，用户向聊天室发送新消息并被添加到聊天内容中时，可以POST给该进程。

现在，我们需要思考一点有关用户界面的问题。用户如何能够向我们的应用程序POST数据呢？这需要一个方法来实现。最简单的方法是当用户要求查看聊天室时，在其发送的页面里创建一个表单。这样，聊天页面顶部会有一个标题，然后，页面会有一个聊天室内容的副本，再然后，在页面底部，要有一个输入框来接收用户名称和用户想要发布的消息。

为了用App Engine实现此应用程序，我们需要建立一个RequestHandler，用来实现聊天室内容的GET操作，然后建立另外一个RequestHandler，用来接收POST操作，并向聊天室添加内容。

聊天室的主页与我们在第2章所使用的代码几乎相同。主要的区别在于，我们需要给它添加一些动态内容，也就是说，我们需要生成已发布的消息文本。因此，我们不能只使用静态的HTML，还需要动态生成一些HTML。作为第一次尝试，我们将创建一个全局变量，它包含一个消息列表。显示页面时，将迭代消息列表，并把它们添加到页面上。

chattwo/chattwo.py

```
class ChatMessage(object):
    def __init__ (self, user, msg):
        self.user = user
        self.message = msg
        self.time = datetime.datetime.now()

    def __str__ (self):
        return "%s (%s): %s" % (self.user, self.time, self.message)

Messages = []

class ChatRoomPage(webapp.RequestHandler):
    def get(self):
        self.response.headers["Content-Type"] = "text/html"
        self.response.out.write("""
           <html>
             <head>
               <title>MarkCC's AppEngine Chat Room</title>
             </head>
```

```
          <body>
            <h1>Welcome to MarkCC's AppEngine Chat Room</h1>
            <p>(Current time is %s)</p>
        """ % (datetime.datetime.now()))
    # 输出聊天信息集合
    global Messages
    for msg in Messages:
        self.response.out.write("<p>%s</p>" % msg)
    self.response.out.write("""
        <form action="" method="post">
        <div><b>Name:</b>
        <textarea name="name" rows="1" cols="20"></textarea></div>
        <p><b>Message</b></p>
        <div><textarea name="message" rows="5" cols="60"></textarea></div>
        <div><input type="submit" value="Send ChatMessage"></input></div>
        </form>
      </body>
    </html>
    """)
```

虽然处理POST是全新的一步，但是webapp框架使其很容易实现。在GET请求的处理程序中，我们实现了RequestHandler的一个子类，该类有一个get方法。对于POST请求，我们遵循相同的模式：实现RequestHandler的一个方法，在这个例子中，是post方法。RequestHandler超类保证当post被调用时，该对象的字段包含我们想从请求中看到的所有内容。为了从生成POST的表单的字段中得到数据，我们只要使用表单中指定的标签，调用该请求的get方法即可。在代码中，我们从POST请求得到用户名和消息，并使用它们；我们会创建一个消息对象，并将其加入全局消息列表中。

chattwo/chattwo.py

```
def post(self):
    chatter = self.request.get("name")
    msg = self.request.get("message")
    global Messages
    Messages.append(ChatMessage(chatter, msg))
    # 将消息添加到聊天室后，我们将重定向到根页面，刷新用户浏览器，显示包含新消息的聊天室
    self.redirect('/')
```

现在，我们需要将它组装成一个应用程序。分为两步：首先，我们需要编写Python代码，创建应用程序对象，并将请求映射到我们编写的请求处理程序上；第二，我们需要编写app.yaml文件。然后，我们就可以测试该应用程序了。

app.yaml文件和之前的几乎完全一样。我修改了这个新示例的Python文件名称，所以需要将app.yaml文件中的script项更改为新名称。

chattwo/app.yaml

```
application: markcc-chatroom-one
version: 1
runtime: python
api_version: 1
```

```
handlers:
- url: /.*
  script: chattwo.py
```

以下是Python的webapp的粘合代码:

chattwo/chattwo.py

```
chatapp = webapp.WSGIApplication([('/', ChatRoomPage)])

def main():
    run_wsgi_app(chatapp)

if __name__ == "__main__":
    main()
```

现在，当我们运行该程序时（和上一章一样，使用`dev_appserver.py`），就得到一个简单的、可正常工作的聊天室。来吧，试一下。它使用`dev_appserver.py`工作得非常漂亮！因此，我们现在可以把它上传到App Engine的服务器上。和之前一样，我们使用`appcfg.py`将其上传到App Engine。

你可以看到图3-2中的结果。它看起来几乎和运行在本地开发服务器上完全一样。我使用两个不同的用户名发送了一些消息，并得到了很好的聊天记录。

图3-2　活动的聊天室应用程序

然后，我需要休息一下，去给我儿子洗澡，并把他送回到床上。回来时，我又发送了另一条消息。你可以从图3-3中看到结果。

当我发送新消息时，所有的旧消息都不见了！聊天记录里除了我刚刚增加的新消息外没有其他内容。我们没有编写任何代码来删除旧消息，事实上，我们的代码根本没有删除消息的功能，我们只是不断地将消息添加到聊天室中。那么，之前的聊天记录发生了什么事？旧消息哪里去了？

图3-3 休息后的聊天室应用程序

它们哪儿都没去，我们遇到了云代码和普通代码的一个基本的、重要的差别。当开发者为自己的服务器编写代码时，知道每个请求都会被自己的服务器处理。如果使用Python解释器，并发送消息给解释器处理，那么，只要开发者需要，Python解释器就在那里。

而当开发者向云服务器发送一条消息，它就会被路由到某个云数据中心的某台服务器上。你无法确保任意两条消息被路由到相同的服务器上，甚至不能保证是同一块大陆的服务器上。即使它们恰好集中在同一台服务器上，也不能保证这台云服务器上的Python解释器会一直运行你的代码。在类似webapp的基于云的编程框架里，请求处理程序是无状态的，也就是说开发者不能指望在处理不同请求时，变量里包含的是之前设定的值。开发者需要在编程时假设每个请求都使用一个全新的Python解释器运行。

该应用程序之所以能够正常工作纯粹是靠运气。当我们在本地运行该程序时，`dev_appserver`只使用一个单一的Python解释器，因此应用程序能在本地运行良好。当我们把它上传到App Engine服务器上时，它就运行在云里。App Engine接收到第一个请求，就开始运行一个Python解释器，并且使用该解释器运行请求。当我提交一个消息，会向App Engine发送第二个请求。主App Engine服务器会看到数据中心处理器上有一个Python解释器，这个解释器刚刚处理了该应用程序的请求，而且解释器并不忙，因此，它会将第二个请求路由到那个解释器。

但后来，我休息了一会儿，并且离开电脑十五分钟。在此期间，App Engine服务注意到运行我的聊天应用程序的Python解释器已经空闲一会儿了，所以它自动关闭了。等我提交接下来的消息时，没有活动的、正在运行聊天室代码的Python解释器，因而App Engine服务启动了一个新的解释器。

要解决这样的问题，当为App Engine构建云应用程序时，就需要明确地知道如何管理在不同请求之间共享的数据。我们不能依赖模块或类变量来管理应用程序的状态：每当我们改变应用程序时，必须显式地存放应用程序的所有数据；而当我们想要访问应用程序时，必须显式地取回数据。

必须牢记的一点是，开发者在云应用中不要臆测基本的数据管理方式，而是需要显式的数据

管理。幸运的是，webapp提供了一种非常好的、持久性服务，称为数据仓库（datastore）。我会在下一章中描述数据仓库。

3.4　参考文献和资源

- RFC 2616：超文本传输协议，HTTP/1.1，http://www.w3.org/Protocols/rfc2616/rfc2616.html。
  W3C的HTTP 1.1协议标准。
- 维基百科关于HTTP的文章，http://en.wikipedia.org/wiki/Http。
  一个简明、全面、非正式的HTTP描述。
- Django，http://www.djangoproject.com/。
  Django是一种广泛使用的网络服务开发平台。它是Google App Engine可供选择的框架之一。很多App Engine的设施都是在Django框架上建模而来，或者是借鉴了Django框架。
- Django Nonrel，http://www.allbuttonspressed.com/projects/django-nonrel。
  Django Nonrel是Django的一个变种，它试图让Django在不基于关系数据库的平台上使用起来更容易，就像App Engine。

第 4 章

云中的数据管理

4

在这一章中，我们来修改聊天应用程序，通过使用Google App Engine的数据仓库来实现持久性存储，在这个过程中了解云应用程序中存储和管理数据的常见问题。

4.1 聊天软件为何不工作？

在上一章结束时，我们的聊天应用程序有一个问题。它似乎是可以工作的，但是如果我们停止使用几分钟，然后再回来，它便丢失了聊天历史记录。

原因是我们的聊天应用程序现在依赖全局变量存储聊天历史记录。但是当我们在云中运行应用程序时，该方法并不可行：全局变量的数据在各个请求之间不一定能够保留。

首先要问的问题是：为什么呢？云应用程序的哪些因素迫使我们要用不同的方式处理数据？

开发者在自己的计算机上运行应用程序时，操作系统创建一个进程。进程分配一些内存用于存储，在内存中存放程序所有的数据。进程继续在计算机上运行，同时，也继续使用着操作系统分配给它的内存，直到开发者退出该进程。如果开发者使用一会儿应用程序，然后离开做些其他的事情，之后再回来继续运行，它仍然是同一个进程，访问着同样的内存。

云则是个完全不同的世界。开发者业已习惯的规则，即使像变量的工作方式这样简单的事情，都有很大的不同。这个程序不是运行在开发者的计算机上，天晓得它运行在哪个或哪些计算机上，这些计算机也不知道位于哪个与网络相连的数据中心内。我们编写了一个云应用程序，并使用Google App Engine的服务将其上传。从我们这样做的那一刻起，程序就在运行，但是它的运行并不像开发者在以往使用桌面计算机时所期望的那样。一个程序在云中“正在运行”可能并不是真正运行于某台计算机上。实际上，在我写到这里时，我已经完成了Python聊天室应用程序的最终版本，并上传到了App Engine——并且，它确实没有运行在任何一台计算机上！（那么，我是怎么知道的呢？因为几周前我上传了该程序，而且我至少一周没有在浏览器上加载过该页面。所以，现在，App Engine服务器绝对没有在任何计算机上加载过它。但是，如果我在浏览器上加载该聊天应用程序的页面，App Engine就会加载它来处理我的请求。）

正如前面章节内容所述，云应用是围绕请求处理构建的。如果没有等待处理的请求，程序就没有必要在任何计算机上运行了；如果有少数请求到来，在一台计算机上运行程序可能就足够了；如果开发者的应用程序得到了Slashdot网站的关注，被它提到过，则可能需要上千台机器来处理

所有收到的请求！对于程序而言，这是云之所以有趣的地方之一：它给开发者提供了一个具有自扩展性的平台。

当然，这是有代价的。如果程序可以在多台计算机上运行，或者它没有在任何一台计算机上运行，那么，开发者就不能确定当代码开始处理请求时，内存里实际存储的是什么。

当程序开始处理请求时，开发者不能依赖于其应用程序的请求处理程序在内存中保留的数据。那么，在开发者处理请求时，必须将每个将来可能要查看的数据显式地保存在一个共享存储区域内，该区域被称为持久性存储。开发者在处理请求时要用到的每个数据必须从持久性存储中取回。

云：函数式程序设计？

云作为一种开发环境，带动了函数式程序设计风格的普及。在函数式程序设计中，开发者不允许存储不稳定状态。换而言之，开发者不能够使用赋值语句修改变量值。因而，开发者不能存放任何在同一个函数的不同调用之间进行共享的数据。开发者必须通过参数显式地传递函数所需要的所有数据。这样的风格对于云编程而言，是非常自然的。

如果开发者不习惯这种风格，可能会认为函数式程序设计看起来很不自然，也非常难以掌握。但它其实并非如此——实际上，一旦习惯了函数式程序设计，它就具有了十足的吸引力！而且，开发者的应用程序越复杂，函数式的风格就越具有吸引力。

在 Google，我们一般使用 3 种语言编程：C++、Java 和 Python。这些都不是函数式程序语言，而都是重状态的、规则式的、面向对象的程序语言。但是我在这种代码基础上阅读和编写的代码越多，就越发现函数式编程是构建大型程序的最好方式。我发现如果代码基本上是函数式的，就更容易理解和测试，而且不太可能产生令人痛苦的错误。这就是为什么当我看到代码不是函数式时会有些畏缩的原因。我编写的几乎所有程序，最后大都是函数式的——现在，我只有在编程语言和编译器不能保持代码效率，无法完成任务目标时，才使用非函数式代码。

我们如何使用永久型存储开展工作呢？这是云编程与老式的本地应用程序的本质不同之处。当开发者开始编写云应用程序时，其过程看起来累赘且令人沮丧。但这并不是一件坏事。事实上，开发者的应用程序由无状态的各部分组成，这会有极好的效果，这是使程序具有可扩展性的关键。如果应用程序每5秒处理一个请求，在一台计算机上将其作为Python应用程序运行，会工作得很好。开发者能够使用全局变量表示数据，并且不需要处理特殊的持久性存储，就能得到所有结果。但是，如果应用的用户越来越多，会发生什么事情呢？每秒1个请求？没问题。每秒10个呢？没问题。每秒100个呢？可能开始有点困难了。每秒1000个呢？10 000个呢？100 000个呢？迟早在某个点，应用程序会崩溃，它无法处理所收到的大量请求。但是在云中，随着请求数量的增加，运行应用程序的机器的数量也能够增加，所以不管用户请求的速度如何，开发者总有能力处理这些请求。一个良好的永久型存储机制意味着开发者不需要关心有多少台机器正在运行其程序，无

论是一台、十台还是一千台机器都没有任何区别。在我自己的工作项目中，我的代码每晚在由数千台机器组成的网络上运行。在那样的环境中，使用全局变量共享数据显然是荒谬的：我的Python程序里对一个全局变量的赋值怎么可以被一千台机器共享？但是因为系统使用了永久型存储，所以这根本不是问题。当系统的一部分变得很慢，并且开始超过其执行期限时，我只需改变一个配置文件，声明它能够使用的最大的机器数量就可以了，而且，这就是我需要做的所有工作。程序开始在更多机器上运行，也会更快地执行完。

云中的通用数据管理

每个云编程系统都提供了某种存储永久型数据的机制。尽管确切的机制不尽相同，但是，基本机制几乎都类似于数据库。有些系统使开发者可以访问小型的、快速的数据库系统（如 MySQL），有些系统则提供了更灵活的、类似数据库的存储方式，如 App Engine。虽然本书中我们只了解 App Engine 的数据仓库机制，但是，持久性存储还有很多其他实现，其中一些也可以用于 App Engine 程序。

4.2　聊天软件的持久性改造

Google App Engine有一个定制的数据持久性系统，称为App Engine数据仓库（App Engine datastore）。App Engine 数据仓库非常类似于数据库，只是对于像Python对象这类数据而言，使用起来更加简单。与关系数据库不同的是，数据仓库不需要严格的数据模式；对于开发者存储和管理持久性数据而言，数据仓库非常灵活，并且是动态的。为了用Python检索并获取数据，App Engine提供了一个自定义的查询语言，称为GQL。GQL看起来和用于查询传统关系数据库的SQL语言非常相似，但是它是为使用数据仓库对象而不是关系表定制的。开发者并非必须使用GQL进行数据仓库相关的工作，但是GQL确实非常适合这项任务，并且易于使用。

GQL 和数据仓库

如果开发者之前使用过关系数据库，就会知道，SQL 查询语言是数据库密不可分的一部分。SQL 是用户与数据库交互的唯一途径。在数据库中，所有的数据都存储在关系表中，用户通过 SQL 实现对这些数据库表的每个操作。正因为如此，对关系数据库编程人员来讲，SQL 和数据库实际上近乎是同一事物。

在 Google App Engine 的数据仓库中，工作方式根本不是这样的。App Engine 提供了一个基本的存储引擎，允许用户存储对象和数据。该存储系统根本没有查询语言。数据仓库只是一个大容量、大规模并行存储系统。

在内部，数据仓库使用索引来协助迅速完成查询工作。通常情况下，在 App Engine 开

发环境中，索引是基于开发者所运行的测试自动生成的。开发者也可以构建自己的索引，我们将在第在内部,数据仓库使用索引来协助迅速完成查询工作。通常情况下,在 App Engine 开发环境中，索引是基于开发者所运行的测试自动生成的。开发者也可以构建自己的索引，我们将在第 14 章中了解该内容。GQL 是一种查询语言，专门用来处理数据仓库和索引。基本上，GQL 就是数据仓库索引的一个封装，它是一个便捷的、轻量级的查询语言，可以毫不费力地查找数据仓库的索引。它并不是数据仓库的一部分，而只是一个帮助开发者使用数据仓库的有效的函数库。

理解了 GQL 的使用目的，我们也就能从中领会到 GQL 的工作原理。GQL 不是数据仓库的组成部分，它只是一个协助用户使用数据仓库索引的工具。

4.2.1　创建和存储持久性对象

数据仓库有很多选项，开发者可以选择最适合自己应用程序的方式开展工作。基本的数据仓库操作很简单，由于开发者对数据仓库的使用越来越多，在需要时，也可以逐渐使用其更多复杂的功能。现在，我们仍然从基础开始。

通常，采用App Engine的数据仓库与采用Python编程有很大的区别。通常情况下，当开发者用Python创建一个类时，并不需要声明类的各个字段，只要赋值，这些字段就自动创建了。而要使用数据仓库，开发者就不得不舍弃一些灵活性。采用数据仓库，开发者必须创建对象的模型（model），告诉数据仓库对象有哪些字段，以及这些字段都有什么类型的值。（实际上，开发者似乎可以采用一种名为expando模型的Python编程风格，该模型可以在需要时再赋值，但是开发者确实不应该使用这种方式。对于云应用程序而言，开发者应该认真考虑自己的数据，并为其定义一个适当的模型。）

背景介绍已经足够了。掌握App Engine数据仓库的最简单方法就是直接深入其中，分析代码。正如前面所说，在数据仓库中，开发者需要定义一个模型，告诉数据仓库各个对象的相关内容。在Python中，模型是一个类对象，它是`db.Model`的一个子类，其中，各个字段是通过创建模型类的类成员来定义的。这是一种笨拙的、非Python的做法。下面，我从聊天应用程序取出`ChatMessage`，并将其转变为一个数据仓库模型：

persist-chat/pchat.py

```
class ChatMessage(db.Model):
    user = db.StringProperty(required=True)
    timestamp = db.DateTimeProperty(auto_now_add=True)
    message = db.TextProperty(required=True)

    def __str__ (self):
        return "%s (%s): %s" % (self.user, self.timestamp, self.message)
```

在App Engine的数据仓库中，一个模型定义了一组命名的特性（properties）集合。开发者通过创建db.Model的子类来定义一种可存储的对象，并且，通过给类本身的类变量赋以适当的类型来定义该对象的特性。数据仓库支持很多数据类型：字符串、数字、日期、列表、引用等。它甚至让开发者定义自己的可存储对象的新类型。我们在第13章中，会进一步讨论开发者可以实现哪些复杂工作。

我们的聊天消息有三个字段：一个包含发送消息的用户名称的字符串、一个包含消息的字符串，以及表明消息何时发送的时间标签。每个字段都被声明为一个特性。

- User　用户名只是一个字符串特性。每条消息都必须有一个用户名，所以我们通过关键字参数required=true指定它不能为空。在数据仓库中，字符串特性的值就是一个Python字符串，不能超过500个字符。
- time　时间特性是db.DateTimeProperty的一个实例，表明该特性的值是Python中datetime的一个实例。对于这个特性，我们可以使用一个有趣的功能，就是数据仓库使用Python类表示特性所用的方法。每条消息都应该有一个时间标签，但是我们并不想在创建一条消息时必须指明时间戳，而是希望时间戳就是现在——也就是说，该消息被应用程序收到的时间。因此，我们为该特性添加了一个特殊的关键字参数auto_now_add，表明"如果在该模型类型的实例被创建时，该特性未被显式地初始化，那么自动将其初始化为当前时间"。因为该特性由Python类的一个实例表示，该类可以被定义为可定制的初始化参数，从而提供了类型特定的功能，如auto_now_add，而不需要任何特殊的原型。在第13章中讨论高级数据仓库的话题时，用户将会了解到如何定义自己的、新的特性类型，并且提供自己的类型特定的扩展功能。
- message　最后，我们来看一下消息的内容。和user字段一样，message也是一个必需的字符串特性。但是在数据仓库里，字符串不能超过500个字符。可能大多数的聊天消息比这个长度短，但并不是所有的聊天消息都小于500个字符。因此，我们没有使用db.StringProperty，而是使用db.TextProperty。这是一个想要多长就可以多长的字符串，但是因为它可以是任意长度，所以开发者不能用它来进行排序或搜索。

因为我们已经通过描述实例所需要的信息创建了一个模型，所以现在不需要提供我们自己的初始化方法了。db.Model会基于我们声明的类的各个字段的特性名称和类型自动生成一个包括关键字参数和类型的初始化程序。

我们已经有了一个可存储类，如何实际存储数值呢？再简单不过了：作为db.Model的子类的实例，每个对象提供了一个没有参数的方法，该方法名称为put。如果开发者调用了一个对象的put方法，这个对象就被存储到此应用程序的数据仓库中。下面是我们POST处理程序的修改版本，它创建了ChatMessage类的一个实例，然后，在msg.put()处，存储了新的聊天消息：

persist-chat/pchat.py

```
class ChatRoomPoster(webapp.RequestHandler):
    def post(self):
        chatter = self.request.get("name")
```

```
        msgtext = self.request.get("message")
        msg = ChatMessage(user=chatter, message=msgtext)
❶       msg.put()
        # 现在我们已将消息加到聊天内容中，然后重定向到根页面
        self.redirect('/')
```

就是这样。在模型实例上调用`put()`将实例存储到数据仓库中，并且使其可用于查询和取回。

4.2.2 取回持久性对象

我们需要知道的最后一件事情就是如何取回所存储的数据。下面是我们的GET处理程序的一部分，用于从数据仓库取回所有的消息；该方法的其余部分，也就是其他的所有代码，取回消息并打印，则一点都没变。

persist-chat/pchat.py

```
# 输出聊天消息集合
messages = db.GqlQuery("SELECT * From ChatMessage ORDER BY time")
for msg in messages:
    self.response.out.write("<p>%s</p>" % msg)
```

开发者可以使用GQL查询语言取回数据。正如代码中所示，GQL和SQL有很多相似之处。最大的区别是GQL不对表查询，它查询的是模型类型。我们的聊天室中的查询，选择的是`ChatMessage`的所有实例，而不是表中的所有行。

根据开发者要查询的内容，有时很明显需要使用不同的GQL风格。开发者通过调用模型类的`gql`方法，可以省略查询中的`SELECT * FROM type`部分。例如，在我们上边的代码中，GQL查询也可以写成`ChatMessage.gql("ORDER BY timestamp")`。

4.2.3 使用GQL查询改进聊天软件

我们的聊天应用程序有一个问题，它很冗长。现在，用户每次刷新聊天显示，就会看到整个聊天内容。当对话已经持续一段时间以后，这一过程就会变得很长，而用户感兴趣的是最近一段时间的那部分聊天内容，也就是总在页面底端的那部分内容。

聊天室的用户不想又不得不经常滚动至他们以前已经看过的消息。大多数时候，他们知道之前说过什么，只想看到最新的消息。例如，他们可能只想看到该聊天室的最后二十条消息，或者只想看到最后五分钟内发布的消息。

使用GQL，在GQL查询语句中增加一些子句修正这个冗长的问题简直轻而易举。要看到最近二十条消息，我们可以添加一个`LIMIT`子句；要看到最后五分钟的消息，我们可以添加一个`WHERE`子句。

当然，我们并不想限制用户，使他们只能看到某个简洁的视图——当他们第一次进入一个新的聊天室时，可能希望看到整个聊天历史记录。因此，我们会分别为这两种新场景，给我们的应用程序添加新的处理程序。我们会保留完整的聊天视图，并给时间限制和计数限制的精简视图增加两个新的URL。

App Engine 数据仓库与关系数据库

讲到这里，数据仓库模型和关系数据库表之间的差异可能听起来并不大。毕竟，每一个 ChatMessage 的实例是完全相同的：它们有一组类型字段，看起来和关系数据库中的列很像。乍一看，数据仓库很像关系数据库，只是其使用程式化的 Python 类创建表，而不是使用 SQL 的 CREATE TABLE 语句。

事实上，情况远非如此。数据仓库中，数据类型和数据结构的范围比关系数据库要丰富得多。在数据仓库中，我们可以有一个包含列表类型特性的模型，列表中的元素可以是任何可存储的值，并且开发者可以将列表的元素作为 GQL 查询的一部分。开发者可以用引用特性来描述对象间的非包含链接，还可以用层次型、树状结构的数据类型，以及遍历树的查询。（这并不是说数据仓库比关系数据库好，它们只是不同。例如，关系数据库的合并操作的性能远远优于数据仓库，但是数据仓库可以让开发者在应用程序中使用熟悉的数据结构，使其条理清晰，且具有可扩展性。）

4.2.4　添加计数限制视图

首先，我们来添加计数视图。GQL查询有一个LIMIT子句，可指定查询结果的最大数目。例如，当用户声明LIMIT 20时，会得到按照指定顺序排序的、与查询结果匹配的前20个值。由于想要得到时间上最近的20个查询结果，需要确保我们想要的结果排在前边。我们按时间排序，最近的消息先显示，就可以实现这个效果了。

计数视图使用RequestHandler实现，除了有两行不同外，其余都和ChatRoomPage相同。因此，我复制了ChatRoomPage，并将副本改名为ChatRoomCountViewPage。修改后的get方法如下所示：

persist-chat/pchat.py

```
class ChatRoomCountViewPage(webapp.RequestHandler):
    def get(self):
        self.response.headers["Content-Type"] = "text/html"
        self.response.out.write("""
           <html>
             <head>
               <title>MarkCC's AppEngine Chat Room (last 20)</title>
             </head>
             <body>
               <h1>Welcome to MarkCC's AppEngine Chat Room</h1>
               <p>(Current time is %s; viewing the last 20 messages.)</p>
           """ % (datetime.datetime.now()))
        # 输出聊天消息集合
❶       messages = db.GqlQuery("SELECT * From ChatMessage " +
                               "ORDER BY timestamp DESC LIMIT 20").fetch(20)
```

```
❷        messages.reverse()
         for msg in list(messages):
             self.response.out.write("<p>%s</p>" % msg)
        self.response.out.write("""
           <form action="/talk" method="post">
          <div><b>Name:</b>
           <textarea name="name" rows="1" cols="20"></textarea></div>
           <p><b>Message</b></p>
           <div><textarea name="message" rows="5" cols="60"></textarea></div>
           <div><input type="submit" value="Send ChatMessage"/></div>
           </form>
         </body>
       </html>
       """)
```

只有以下两处真正的变化。

❶ 在查询语句中，我们指定排序方式为降序，所以发布到聊天室的最近20条消息会排在查询结果的最前边（ORDER BY time DESC），同时，我们将查询结果限制为20个（LIMIT 20）。

❷ 查询语句生成了按照时间降序排列的消息，最近的消息排在最前边。然而，当我们的用户读取聊天记录时，这仍不是他们希望看到的顺序：当用户查看聊天记录时，希望聊天记录按照自然顺序出现，也就是说，最近的消息应该显示在最后。所以，我们需要在打印出来之前将查询结果反序。

4.2.5　添加时间限制视图

增加基于时间选择一部分聊天记录的视图要比增加计数限制的视图更为复杂。它需要在查询中增加比较，并且遇到了GQL最大的限制之一：在GQL查询中，开发者不能做任何计算，不能使用如x+1这样的表达式。每个计算都需要用Python代码在查询语句之外完成，然后再插入到查询中。

此外，开发者可以使用参数的地方是有限的。一般情况下，开发者只能在查询语句的WHERE子句中使用参数。

要想真正理解这两个限制的含义，我们需要看看GQL的一些参数。一个参数基本上是查询语句里的一个槽，我们可以在槽里注入一个Python值。

例如，如果我们想让计数有限的视图支持不同数量的消息，在查询语句中使用参数看起来就很自然：`ChatMessage.gql("ORDER BY time DESC LIMIT :1". 20).:1`是一个查询语句的参数，将被查询字符串后跟着的第一个未命名的参数替换，即本例子中的20。不幸的是，我们不能这样做，开发者在LIMIT子句中不能使用参数。然而，我们仍然可以使用Python的字符串完成替换这个工作：`ChatMessage.gql("ORDER BY time DESC LIMIT %s" % 20)`。

我们可以在时间限制视图中使用参数是因为相关参数是WHERE子句的一部分。要实现时间限制视图，必须进行一些时间运算。如果要显示最后五分钟内发布的消息，我们会在查询中将其描述为类似以下的内容："时间标签大于当前时间减5分钟的所有消息。"

在Python中，我们可以用datetime模块：`datetime.now() - timedelta(minutes=5)`

来表示“当前时间减5分钟”。要在查询语句中使用，我们只需要注入一个参数。因此，我们编写为：

```
ChatMessage.gql("WHERE timestamp > :fiveago ORDER BY time",
                fiveago=datetime.now() - timedelta(minutes=5))
```

这就是要做的全部事情。只需复制ChatRoomPage，将其重命名为ChatRoomTimeViewPage即可，并用上述代码片段替换查询语句，这样就实现了时间限制视图。

参数可以是编号或命名。如果是命名，在Python调用中使用命名参数指定它们的值，如同我们上边做的那样。如果是编号，开发者只需要按顺序指定各个参数。例如，我们可以在时间限制视图查询语句中使用编号参数，如ChatMessage.gql("WHERE timestamp > :1 ORDER BY time", datetime.now() - timedelta(minutes=5))。对于任何给定的查询语句，最好全都使用命名参数或者全都使用位置识别的参数，混合使用这两种方式则容易引起困惑。（事实上，我认为大多数时候，开发者应该完全使用命名参数。这样做唯一的缺点是会稍微多打点字，但是它使代码更清晰，而且出错时，也更容易发现。）

当然，要能看到该视图并对其进行测试，我们需要修改WSGIApplication，将查询指向我们的两个限制视图。现在，我们的应用程序有三个视图：完整会话视图、时间限制视图和计数限制视图：

persist-chat/pchat.py

```
chatapp = webapp.WSGIApplication([('/', ChatRoomPage),
                                  ('/talk', ChatRoomPoster),
                                  ('/limited/count', ChatRoomCountViewPage),
                                  ('/limited/time', ChatRoomTimeViewPage)])
```

我们还没有在视图间切换的好方法，而且各视图的实现有过多数量的冗余。我们会在后续章节中学习如何消除冗余。但到此为止，我们已经有了可运行的程序。

4.3　参考文献和资源

- Python数据仓库API，http://code.google.com/appengine/docs/python/datastore/。
  官方的Google App Engine数据仓库文档。

第 5 章

Google App Engine的登录认证服务

开发者用App Engine构建的大多数网络应用程序必须要能够跟踪用户的使用。开发者需要能够让用户进行登录，并根据获得的权限执行操作。在本章中，我们将了解如何管理用户，以及跟踪哪个用户发出了哪个请求。为了做到这一点，我们将使用一种App Engine的服务，这种服务是独立于webapp框架的一部分App Engine API。

5.1 users 服务简介

我们的聊天系统已经有了一个很好的开端，但它的功能还是非常有限。我们开始时设计了支持多个聊天室的蓝图，不幸的是，现在还不能很好地实现那个设计。在那个设计中，一个给定的用户可以订阅多个聊天室，而且聊天应用程序可以追踪到不同的用户分别订阅了哪些内容。但是目前，我们的聊天应用程序没有任何方法能跟踪到是谁发出了特定的请求，因此我们也不知道显示给用户的是哪个聊天室。

登录和认证不仅是聊天系统这样的应用程序需要，可能在每个App Engine应用程序开发中都需要考虑。

像登录这样的功能无处不在，而且是非常必要的，几乎会出现在每个应用程序中，App Engine提供了所谓“服务”的API。服务就是由App Engine实现所提供的模块，无论开发者使用什么样的框架构建应用程序，其App Engine应用都可以使用该模块。即使开发者决定在应用中使用Django，也仍然可以使用所有的App Engine服务。

设置认证最简单的方法是使用Google App Engine的users服务，该服务在Google账户上实现。如果开发者的应用使用Google邮箱地址作为主要的标识符，那么该应用就可以使用Google的登录服务。如果开发者愿意，可以构建自己的认证服务，或者使用第三方认证服务，没有什么理由强迫开发者必须使用GAE的登录服务。但是GAE的登录服务非常方便，而且对大多数应用程序而言，它可能就足够了。使用其他服务不会有很大的不同——基本的机制都是相似的。

5.2 users 服务

Google的登录由users服务实现。users服务跟踪当前登录的用户，并提供给应用程序登录和注销页面的功能。

5.2.1 用户对象和当前用户

使用users服务可以做的最简单的事情就是从当前登录的用户中取回user对象。开发者可以随时调用users.get_current_user()取回用户。如果没有用户从那个客户端登录，调用返回None；如果有一个登录的用户，开发者会收到一个Python对象，该对象有以下三个实例方法。

- nickname()　与用户相关的文本名称。这最终是用户可配置的，但此刻，它只返回电子邮件地址中@符号之前的部分。我这里就是markcc。
- email()　用户的电子邮件地址。这里是markcc@gmail.com。
- user_id()　用户的永久标识符。请把它当作一个不透明的字符串。用户随时可以修改他们的邮件地址或者昵称，但是他们的user_id始终都是相同的。该字符串并不用于显示目的，但是如果开发者想要记录用户的永久信息——如一个用户在电子商务网站的一组订单——就可以使用user_id作为标识符，而不用管它的内容到底是什么。

5.2.2 用户登录

我们需要为用户提供一个登录页面。幸运的是，我们不需要设计自己的登录页面——毕竟，每个登录页面都大同小异。users服务提供了自动生成登录页面的机制。

标准的users登录服务的目的是作为一个桥梁：那就是一个页面，用户沿着这个页面到达他们真正想要访问的内容。在我们的聊天应用程序中，用户首先会看到一个欢迎界面，他们从这个欢迎界面进入聊天室，查看正在进行的会话。但是，我们想让用户在访问聊天室前先登录，以使我们知道他们都是谁。如果用户没有登录就想要进入聊天室，则会被导向登录页面。只要他们完成登录，就会被直接导向聊天页面。

考虑到这一点，users服务工作的方式就是处理用户登录，并且由开发者提供给它一个目标页面。但是，当用户成功登录后，它会自动将用户重定向到目标页面。

我们的聊天程序就具有这种非常典型的风格。我们可以在聊天页面的RequestHandler的get方法中增加一些逻辑操作，使它以这种方式工作。

在get的开头，我们调用users.get_current_user()检查用户是否已经登录。如果返回的是一个已登录的用户，我们就继续操作，显示页面；如果不是，则创建一个登录页面，将聊天页面本身作为成功登录后的重定向页面。这听起来有点让人害怕，但其实没有听上去这么复杂：允许用户登录，并且如果他们成功登录后重定向到聊天页面的调用方法是：

```
self.redirect(users.create_login_url(self.request.uri))
```

换句话说，我们正在执行一个重定向，告诉App Engine将用户导向登录页面。登录页面通过

users服务调用users.create_login_url生成。当用户成功登录后，我们将其重定向到聊天页面。我们甚至不需要记住聊天页面的URL，仅使用了原来请求的URI，该URI在请求处理程序self.request.uri中可以得到。

5.3 整合 users 服务到聊天软件中

现在我们知道如何登录了，可以将users和登录整合到聊天应用程序中。要做到这一点，我们需要对聊天应用程序做一系列的修改：

(1) 修改聊天页面，要求用户登录；

(2) 使用登录的user对象设置聊天消息的user字段，并从聊天信息输入表单中删除用户字段；

(3) 修改POST消息处理程序以使用已登录的用户。

我们已经在5.2.2节中，看到了如何完成用户登录。只需更多类似的代码，就可以将该功能集成到我们的聊天页面请求处理程序中。聊天页面请求处理程序的另一个变化是从表单中删除用户名称字段。更新的请求处理程序如下：

login-chat/pchat.py

```
class ChatRoomPage(webapp.RequestHandler):
    def get(self):
      user = users.get_current_user()
❶      if user is None:
        self.redirect(users.create_login_url(self.request.uri))
      else:
        self.response.headers["Content-Type"] = "text/html"
        self.response.out.write("""
          <html>
          <head>
            <title>MarkCC's AppEngine Chat Room</title>
          </head>
          <body>
            <h1>Welcome to MarkCC's AppEngine Chat Room</h1>
            <p>(Current time is %s)</p>
          """ % (datetime.datetime.now()))
        # 输出聊天信息包
        messages = db.GqlQuery("SELECT * From ChatMessage ORDER BY timestamp")

        for msg in messages:
          self.response.out.write("<p>%s</p>" % msg)

        self.response.out.write("""
           <form action="/" method="post">
           <p><b>Enter new message from: %s
           <p><b>Message</b></p>
           <div><textarea name="message" rows="5" cols="60"></textarea> </div>
           <div><input type="submit" value="Send ChatMessage"/></div>
           </form>
         </body>
```

```
            </html>
❷           """ % user.nickname())
```

登录代码在1处，并且完全遵循我们前面描述的模式。如果用户已经登录，聊天应用程序为他们显示聊天内容；如果他们没有登录，将被重定向到登录页面，登录后再返回到聊天页面。

另一处更新在2处，我们从表单中删除了用户名输入字段，用一个提示行代替，提示行使用从当前登录用户对象中得到的昵称。

POST处理程序的修改完全相同。我们只需增加一个get_current_user行的副本，在创建ChatMessage的调用中使用，如下所示：

login-chat/pchat.py

```
    def post(self):
          if user is None:
        self.redirect(users.create_login_url(self.request.uri))
❶     user = users.get_current_user()
      msgtext = self.request.get("message")
      if user.nickname() is None or user.nickname() == "":
        nick = "No Nickname"
      else:
        nick = user.nickname()
      msg = ChatMessage(user=user.nickname(), message=msgtext)
      msg.put()
      sys.stderr.write("****** Just stored message: %s" % msg)
      # 已经将消息添加到聊天室，然后重定向到登录页面
      self.redirect('/')
```

现在，我们已经有提供登录服务的能力，可以开始将聊天应用程序变得更有趣了。我们可以构建很多功能，例如多聊天室、订阅以及其他功能，如连接某些用户、管理用户间私人消息等。当然，天下没有免费的午餐，当我们增加这些功能时，用来显示用户界面的HTML就变得更加复杂了，需要不断地生成相同的HTML代码样板。这项工作费力、容易出错而且令人厌烦。在下一章，我们将学习如何在应用程序中添加订阅和多聊天室功能，以及如何使用模板生成用户界面的HTML，使HTML没有错误、不一致或者其他问题。

第 6 章

代码组织：分离用户界面和逻辑

到目前为止，在我们写的所有代码里，一直通过输出HTML的Python代码来显示用户界面。这样做很费力、很笨拙，而且容易出错。在本章中，开发者将学习模板，用模板来组织Google App Engine应用程序的代码，从而将显示用户界面的代码与应用程序的逻辑实现分离开来。使用模板，开发者可以直接将用户界面作为标记的HTML表单进行编写，而不用通过Python输出语句来显示HTML用户界面代码。应用程序逻辑仍然使用Python，只是在需要生成页面时调用模板即可。

6.1 模板入门

回想第3章“第一个真正的云应用程序”，我们勾勒出一个非常高级的聊天系统。我们设计的聊天系统可以支持多个发生在不同聊天室的并发聊天，并且，用户可以通过订阅不同的聊天室，同时参与多个聊天。

但是，到目前为止，我们构建了只支持一个聊天室的应用程序。在上一章中，我们考虑了完整的聊天应用程序的一个方面：识别登录的用户。我们需要登录功能，既能减少用户每次登录时输入用户名的麻烦，也可以追踪到谁参与了哪个聊天。虽然我们有跟踪用户的能力，但是到目前为止，它只是为用户提供了方便。我们没有管理多个聊天内容，也没有跟踪或保护用户的任何信息，而这些都是我们所需要的。

我们要开始实现一些更有趣的功能。为了实现这些功能，将先介绍一些新的视图。正如在前面的章节中看到的，添加新的视图很痛苦，因为编写代码生成HTML非常笨拙，而且也容易出错。

为了使这些工作更容易、更易于维护，我们将学习使用*模板*（Template）工具。模板为创建描述用户界面的HTML代码提供了一个灵活的、易于使用的系统。它之所以灵活、易于使用，本质上是因为它将系统的逻辑从外观中分离出来。逻辑继续使用Python语言编写，而外观，我们将采用有注释的HTML模板文件。

有很多模板语言。Google App Engine的webapp框架包括了来自开源Django项目的模板，因此，Django就是我们要用的模板。如果开发者喜欢其他模板，可以自行采用：只需要把Python文件放到相应的项目里，它就可以工作了。

6.1.1　为什么学习另一种语言

与所有网络应用程序一样，云应用程序通过生成浏览器中可显示的HTML来构建用户界面。这样做很方便，因为可以很容易地动态创建各种有吸引力的界面。对于创建用户界面来说，网络浏览器是一种非常灵活、一致且友好的平台，开发者通过浏览器就可以使用HTML语言所提供的一切优势。尤其是采用即将出台的HTML5标准，开发者可以创建更加漂亮的界面。

问题是正确地生成HTML将会很难，而且很容易出错。HTML有一套非常详细的语法，并且和大多数编程语言一样，使用某些相同的引用字符。这意味着开发者必须注意代码的编写方式——但无论多么仔细，仍然很容易犯错。

模板有助于解决这个问题。采用模板，就可以将代码分为两部分：计算部分和界面部分。计算部分是用于完成工作、生成想要在网络页面中显示的数据的部分。在计算部分，主要是操作数据——有时可能将其中一些数据通过HTML显示，但是在大多数情况下，计算部分并不显示。

界面部分全是有关生成HTML的。它获得计算部分产生的数据，将其以HTML方式显示。在大多数应用程序中，HTML代码的很大一部分是固定的。在我们的聊天应用程序中，每个页面的显示都必须生成基本的HTML结构，以及页面的头部、消息入口表等等。只有消息列表的内容是变化的，其他一切都是完全相同的。这意味着有大量的`print`语句并没有做任何有用的工作，只是输出固定的字符串以形成HTML页面的一部分。HTML嵌入在代码中，这种嵌入方式确实很难操作。

有了模板之后，开发者用Python（或者其他任何语言）编写代码的计算部分。如果开发者需要将任何HTML显示为计算的一部分，那么，使用嵌入在Python中的HTML字符串即可。至于代码的界面部分，可直接用HTML语言编写，然后使用特殊的元语法嵌入所有需要的计算，这些计算将在HTML文件中插入动态内容。

事实上，开发者可以使用模板编写整个应用程序。但是，正如采用标准的Python代码显示HTML一样，尝试在模板内部使用复杂的编程逻辑也是很困难、痛苦而且容易出错的。

6.1.2　模板基础：采用模板显示聊天软件

使用模板可以做的第一件事是将从数据仓库中取回聊天消息的逻辑，与为显示聊天内容而呈现的页面的逻辑两者分离开来。为实现该功能，我们将应用程序的聊天页面编写为模板，然后修改Python代码来使用该模板。我们用聊天显示页面编写的模板的一个非常简单的版本如下所示：

template-chat/chat-template.html

```
<html>
  <head>
    <title>{{ title }}</title>
  </head>
  <body>
❶   <h1>Welcome to {{ title }} </h1>
❷   <p> Current time is {% now "F j Y H:i" %}</p>
```

```
❸       {% for m in msg_list %}
❹        <p> {{ m.user }} ( {{ m.timestamp }} ): {{ m.message|escape }} </p>
        {% endfor %}
        <form action="/talk" method="post">
          <p><b>Message</b></p>
          <div><textarea name="message" rows="5" cols="60"></textarea></div>
          <div><input type="submit" value="Send ChatMessage"/></div>
        </form>
      </body>
    </html>
```

Django模板是一种嵌入了特殊的非XML语法标记的文本文件。现在，我们要使用它生成HTML，但Django模板并不局限于生成HTML。开发者可以使用Django生成任何由文本文件构建的内容：CSS、XML，甚至是Python代码！

Django模板系统使用大括号标识它的语法。在我们的例子中，模板文件的大部分内容都只是普通的XML。大括号里的内容在模板实际使用时会被替换掉。在第一个例子中，我们只用了几组Django语法的元素：

❶ 我们使用的第一个模板结构是一个变量的引用。在Django中，变量的引用采用两边带双括号的标识符编写——因此，`{{ title }}`是对一个名为`title`的变量的引用。当使用该模板时，它会被命名变量的内容所替换。变量引用也可以使用点号标识符，点号后边的部分是对Python对象字段的引用。我们稍后会看到一个这样的例子。

❷ 接下来我们看看Django如何调用一个标签。Django中的标签像普通编程语言中的函数调用。标签调用被包含在{%和%}之间。调用中的第一个字是标签的名称，其余的是参数。在这个例子中，我们要插入当前时间。Django提供了一个名称为now的标签，该标签会被当前时间替代。Django采用了一系列参数来指定如何格式化时间：

- `F` 当前月份的文本名称；
- `j` 月的天数；
- `Y` 年的数值表示；
- `H` 当前的小时；
- `i` 两位数字格式表示的当前分钟。

开发者在日期格式中还可以使用其他字符——请查看Django模板文档的完整列表。任何不在now格式字符串的字母都包含在替换文本中，所以传给now的参数中的空格和冒号都会包含在文本中。因此，日期显示为这种格式：`Jul 19 2009 02:45`。

❸ 这里，我们使用了一个for循环实现Django标签。在Django模板文件中，标签可以有主体。把它看做与XML类似：XML中，所有的标签可以有一组称为*属性*（attributes）的参数，属性是标签本身的一部分。更复杂的标签也可以有*内容*（content），内容是一些文本和HTML标签的混合，内容嵌入在复杂标签的开始和结束之间。Django模板的标签也是类似的：简单的标签占一行，并通过放在标签内部的参数定义所有内容；更复杂的标签有内容，内容是标签和其对应结尾之间的所有内容。在Django中，主体结束的标志是{%endTAG%}。

6

在这个例子中，标签是一个循环。当该模板被调用时，程序将循环处理指定列表中的每个值，并且为列表中的每个元素生成该标签主体的一个副本。这个循环遍历聊天室的消息：对每一条消息，它会生成一个主体的副本，该主体的变量用聊天室列表中的不同消息替换。

❹ 任何时候，当开发者从一个用户那里得到输入，然后将输入插入到用户界面视图中时，开发者需要谨慎一些，我们将在第17章详细讨论这个问题。如果没有足够小心，恶意用户通过在他们的输入中插入Java脚本，使用开发者的应用程序进行一切恶意行为都是非常容易的事情。要处理该问题，只需在Django模板中通过escape过滤器运行用户输入，即使用|escape。

现在，我们有了用户界面页面的模板，下面需要在Python代码中调用该模板。使用该模板的ChatRoomPage更新版本如下所示：

template-chat/tchat.py

```
class ChatRoomPage(webapp.RequestHandler):
  def get(self):
    user = users.get_current_user()
    if user is None:
      self.redirect(users.create_login_url(self.request.uri))
    else:
      self.response.headers["Content-Type"] = "text/html"
      messages = db.GqlQuery("SELECT * From ChatMessage ORDER BY timestamp")
      template_values = {
        'title': "MarkCC's AppEngine Chat Room",
        'msg_list': messages,
        }
❶     path = os.path.join(os.path.dirname(__file__), 'chat-template.html')
❷     page = template.render(path, template_values)
      self.response.out.write(page)
```

使用一个模板时，开发者需要得到模板文件的路径名，然后调用template.render。

获取路径名需要使用一点Google App Engine中的技巧。正如我们所看到的，在云中，开发者对应用程序环境的控制能力比传统应用程序的小。应用程序和数据就在某个地方，但开发者却不知道在哪里。由App Engine服务器来决定把开发者的数据存放在哪里，而且在没有警告的情况下，它可以随时移动数据。因此，为了访问一个文件，开发者需要向App Engine询问包含你的应用程序文件的目录被存放到了哪里。App Engine使用Python元变量__file__作为当前模块的源文件的引用。App Engine的Python环境保存了文件之间的基本目录关系，因此，如果数据文件与开发者本地开发环境的Python源代码文件放在相同的目录下，那么，该文件与在App Engine服务器上的代码也会存放在相同的目录下。所以，即使开发者不知道App Engine存放其部署代码的目录全名，也可以找到数据文件，因为它与部署代码放在相同的目录下。为了找到模板文件，我们通过__file__元变量，使用标准的Python方法，得到包含我们的Python代码的目录。和其他文件类似，os.path.dirname(__file__)会告诉我们目录，然后我们将其与模板名称组合就得到了模板的路径名。

一旦得到了模板文件的引用，就可以调用template.render。传递给该调用的第一个参数是

模板，第二个参数是一个Python字典。Python字典中的键将成为在模板中可以被访问的变量。我们在模板中引用了`title`和`msg_list`，而这些都是我们加到字典中的键。

6.2　用模板创建相关视图

当开发者构建一个像我们的聊天系统一样的应用程序时，通常会有多个看起来相似但又不完全相同的视图。例如，我们希望有多个聊天室，每个聊天室一个视图。此外，除了每一个不同的聊天视图，我们还需要一个索引视图来选择聊天室。索引视图和聊天视图应该有类似的外观；聊天视图除了有聊天室的名称外，其余应该是与索引视图相同的。

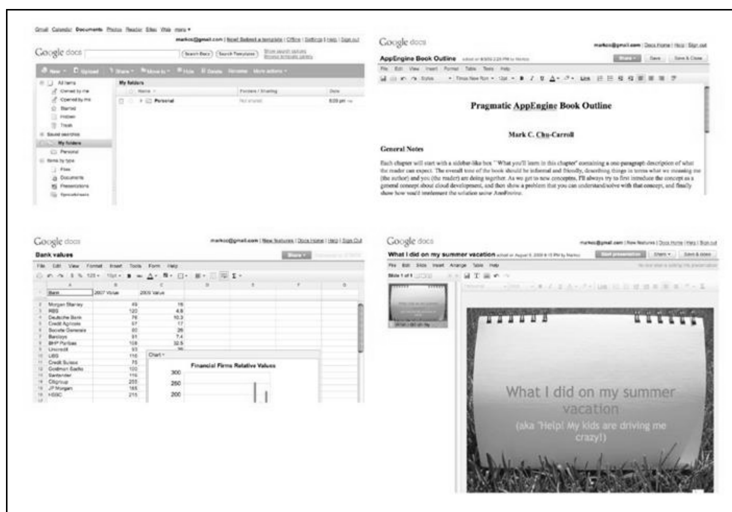


图6-1　Google Docs的通用风格

几乎每一个好的网络应用程序，其所有页面都有一个统一的外观和感觉。一个好的网络应用程序不仅仅是网络页面的集合，它还是一个向用户提供服务的有凝聚力的程序。组成网络应用程序的每个页面提供一部分功能。该应用程序展现给用户的所有页面都有着与其他程序不同的外观，并且它的所有部分都采用此外观，这使用户感觉他们是在使用一个一致的、设计良好的应用程序。例如，图6-1中的Google Docs缩略图。Docs表现为一个办公套件，它启动时的初始化视图看起来像Windows的文件浏览器。当用户打开文档时，会打开一个新的浏览器窗口，以使用户在此文档上工作。每个Docs窗口的基本布局相同，左上角都有Google Docs的标志，右上方有一组控制链接，然后是文档标题和带有一系列蓝色阴影的装饰。用户一看就知道窗口包含的是Google Docs的内容，因为看起来像Google Docs。

通用页面布局和共享风格结合起来，决定了应用程序的外观和感觉。风格由层叠样式表（CSS）定义。基本结构和共享的层叠样式表都可以定义为模板，然后，各个页面可以修改细节，使用子模板来满足其特殊的需求，并且保持应用程序所有内容共享的主模板的默认设置不变。

副本的问题

我最近遇到了一个有关副本的问题。我写博客的网站进行了一次系统升级。在升级之前，所有 URL 的基本路径都是 http://.../cgi-bin/MT/；升级后，基本路径被改为 http://.../mt/。最终发现，很多地方的基本路径都是硬编码的。在升级过程中，其中大多数得到了更正，但还是有一些被漏掉了。结果，文章、评论和管理链接都无法正常工作，需要进行修正。他们花了整整两个星期追查了所有副本，才使一切正常工作。在这两个星期内，该网站对于作者和读者来说都是一团糟。如果他们编写一个计算 URL 的简单函数，在需要生成 URL 的时候就调用该函数，那么他们只需要修改代码中的一行就可以，而且，什么问题都不会发生。

不要让这种事发生在你身上：重用代码，而不是复制代码！

这些视图会有相同的特征，如标题、导航栏、标志等需要显示在所有页面上的内容。开发者也可以使用格式化的元素，如特殊字体、文字样式和配色方案，使应用程序的页面共享同一种外观和感觉。所有的元素都是不同视图共享的。每个页面有自己的特殊内容，以规定页面的功能，但是，所有页面的风格都是一样的，因为都来自相同的代码。

有一个基本的编程规则：在多处存放同一事物的多个副本不是一个好主意。最终开发者总会需要改变一些东西，这时，就很容易漏掉某个副本。我们真的希望能够将所有页面的通用部分只实现一次，然后在特定的页面中引用它们。

6.2.1　模板继承

Django模板最强大的功能之一就是解决了模板继承（Template Inheritance）问题。开发者可以将其网站中所有页面广泛使用的通用结构定义为主模板，然后将各个页面作为该模板的变种。开发者甚至可以创建完整的模板层次结构，这些层次化的模板一层比一层更加特定化。

我们来利用一下模板的优势。我们创建一个主页面布局，顶部有标志以及一些可定制部分。聊天应用程序的主模板如下所示：

multichat/master.html

```
<html>
    <head>
        <title>{{ title }}</title>
    </head>

    <body>

        {% block navbar %}
            <h3>Available Chats</h3>
        <div id="navbar">
          <ul>
```

```
        {% for c in chats %}
        <li><a href="/enterchat?chat={{c}}">{{ c }}</a></li>
        {% endfor %}
      </ul>
    </div>
    {% endblock %}

    <h1>Welcome to {{ title }} </h1>
    <p> Current time is {% now "F j Y H:i" %}</p>

    {% block pagecontent %}

    <p> This is template text. If you're seeing this in a page rendered
      by chat, something is wrong.</p>

    {% endblock %}

  </body>
</html>
```

主模板不能被直接使用。如果直接使用，会生成一个包含以下消息的网络页面："这是模板文本。如果您在聊天室显示的页面中看到该消息，那就是出错了。"主模板提供了一个构建其他模板的基本格式。

基本页面布局和我们一直使用的页面布局大体相同。用户唯一可见的修改是我在页面顶部增加了一个新标志。同时，我也在模板中增加了block标签。block标签识别模板内容中可以被子模板替换的部分。我们使用master.html作为主聊天应用程序的模板。要得到聊天应用程序的基本视图，并用新的主模板来构建该视图，我们需要创建一个新的子模板，如下所示：

multichat/basic.html

```
❶ {% extends "master.html" %}

❷ {% block pagecontent %}
  <p> All chat messages as of {% now "H:i" %}</p>

  {% for m in msg_list %}
  <p> {{ m.user }}@{{ m.timestamp }}: {{ m.message|escape }} </p>
  {% endfor %}

  {% endblock %}
```

我们通过extends标签将其声明为子模板，该标签必须在文件的开头。然后在我们想要改写的地方放置一个block标签。其结果会是一个包含master.html内容的模板，但是其pagecontent块会被我们之前使用的消息显示循环所替换。

到目前为止，模板看起来很好，因为它将HTML代码与Python代码分离开来，为我们提供了一种清晰的编码思路。我们能够分离用户界面和应用程序逻辑，只需要设置好它们之间传递的参数即可。但这只是我们看到的使用模板所做的工作的一个表面现象！模板继承是一个难以想象的、实用的、强大的机制。我们在本书的后续部分会一直使用模板，并用其为应用程序创建更好的界面。

6.2.2　使用模板定制聊天视图

让我们使用所学的模板知识，把聊天应用程序变得更好些。我们可以做的就是改进最近20条消息的视图。在当前的版本中，发送消息时显示的日期和时间的形式是非常繁琐的。但由于大多数时候我们是显示一个活动的聊天室内容，所以其时间戳中的大部分文本是相同的。这样显示不仅浪费空间，迫使消息占用了更多的显示行，而且逐渐使得内容越来越难于阅读。在理想情况下，我们不需要包含全部时间戳，只要一些时间表示。我们可以修改最近20条消息视图的消息显示，使其不显示完整的时间，只标记出每条消息从发送后经过的时间。

在计算部分，我们需要修改Python代码，计算经过的时间，并将这个时间加到我们传递给用户界面模板的消息上：在用户界面代码中，我们需要创建一个新的、修改后的showmessage块，以显示消逝的时间，而不是时间戳。

首先，我们需要更新应用程序的计算部分，增加时间戳。这实现起来很简单，只要使用Python标准的timedelta类就可以：

multichat/tchat.py

```
class ChatRoomCountedHandler(webapp.RequestHandler):
  def get(self):
    user = users.get_current_user()
    if user is None:
      self.redirect(users.create_login_url(self.request.uri))
    else:
      self.response.headers["Content-Type"] = "text/html"
      messages = db.GqlQuery("SELECT * From ChatMessage ORDER BY timestamp ""DESC LIMIT 20")
      msglist = messages.fetch()
    for msg in msglist:
      msg.deltatime = datetime.datetime.now() - msg.timestamp
      template_values = {
        'title': "MarkCC's AppEngine Chat Room",
    'msg_list': messages.fetch(),
        }
    path = os.path.join(os.path.dirname(__ file__), 'count.html')
    page = template.render(path, template_values)
    self.response.out.write(page)
```

此代码中唯一有修改的地方就是在GQL查询语句后增加了循环，循环中给该消息对象增加了timedelta字段。这一小段代码需要关注的地方是：我们在数据仓库对象中增加了一个字段，但由于它不是我们声明的数据仓库特性，对存储对象不起任何作用。即使我们对增加了时间差的消息调用put，也不会改变存储对象的任何内容。

现在我们已经更新了应用程序中的计算部分，程序会计算和存储自消息发送后的时间，那么，我们还需要通过替换showmessage块来更新界面。可创建如下代码实现该功能：

multichat/tchat.py

```
❶ {% extends "master.html" %}

❷ {% block pagecontent %}
  <p> Last 20 messages as of {% now "H:i" %}</p>

  {% for m in msg_list %}
  <p> {{ m.user }}: {{ m.message|escape }} ({{ m.deltatime }} seconds ago)</p>
  {% endfor %}

  {% endblock %}
```

❶ 我们首先声明这是master.html的一个模板扩展，这意味着生成的页面除了我们特意重写的块之外，一切都与master.html相同。

❷ 将主页面中的pagecontent块重写，替换掉原来的块，从而呈现出聊天消息。

6.3 多聊天室

既然我们已经知道如何使用模板，就可以很容易地将各种不同的视图很组合在一起了。我们使用这些知识修改聊天应用程序，使其更加实用。在原始的应用程序架构中，我们希望支持具有订阅功能的聊天室。下面，我们先开始提供多聊天室的功能，在后续部分中再考虑订阅问题。

6.3.1 更新多聊天室的逻辑

既然我们想支持多聊天室的功能，就需要用一种方法来保存可用的聊天室列表。稍后，我们会增加一个管理视图，用于管理该聊天室列表。但现在，我们先用硬编码来实现它。在此，不需要考虑更新不一致的问题，因为它永远不会被更新，每次聊天消息初始化时它都会被重设为相同的值。

另外，我们给聊天消息增加一些内容，以便于知道它们属于哪一个聊天室。这个实现很简单：只要在类中增加一个数据仓库的字段。我们还需要修改POST处理程序，让它设置该聊天室字段，在设置聊天页面时，我们会看到这是如何实现的。修改后的ChatMessage和硬编码的聊天列表如下所示：

template-chat/tchat.py

```
class ChatMessage(db.Model):
    user = db.StringProperty(required=True)
    timestamp = db.DateTimeProperty(auto_now_add=True)
    message = db.TextProperty(required=True)
    chat = db.StringProperty(required=True)

CHATS = ['main', 'book', 'flame' ]
```

我们还需要修改创建聊天消息的POST方法，使其设置聊天消息的聊天室字段。我们在请求中增加一个字段，触发该POST方法。在POST处理程序中，我们获取请求的chat字段，并将其添加到聊天初始化程序中。修改后的POST处理程序如下所示：

template-chat/tchat.py

```
class ChatRoomPoster(webapp.RequestHandler):
  def post(self):
    user = users.get_current_user()
    msgtext = self.request.get("message")
    msg = ChatMessage(user=user.nickname(), message=msgtext, chat="chat")
    msg.put()
    # 我们已经将信息添加到聊天室中，下一步将重定向到根页面
    self.redirect('/')
```

6.3.2　构建多聊天室的登录页面

当有人第一次访问聊天应用程序时，我们需要提供一个登录页面——一个通用的前端页面。对我们的应用程序而言，前端页面将所有聊天室中最近的二十条消息显示给用户。从该页面中，用户可以看到哪个聊天室是活跃的，然后从导航栏中选择一个。用户不能从登录页面发布消息，因为他们还没有选择聊天室，我们也不知道将消息发布到哪个聊天室。

现在，我们将填充第一个真正显示的页面——登录页面。我们不改变工具栏。当前，我们先基于之前的版本创建一个简单的页面，如下面的HTML所示：

multichat/landing.html

```
{% extends "master.html" %}

{% block pagecontent %}

{% for m in msg_list %}
 <p> <b>({{ m.chat }})</b> {{ m.user }} ( {{ m.timestamp }} ):
 {{ m.message|escape }} </p>
 {% endfor %}

{% endblock %}
```

这正是在6.1节使用的原始的聊天室模板中的主体，只增加了一点：聊天消息在行的开始处，用黑体显示它所属的聊天室。

为了显示此信息，创建了一个新的请求处理程序。该程序和之前的请求处理程序很像，例如我们在前面ChatRoomCounted中使用的那个处理程序。

然而，这个新的聊天登录页面有一些不同的地方，它有导航条：通过聊天室列表，用户可以从中选择聊天室。当用户点击其中的链接时，就会被导向一个特定的聊天室。接下来我们要做的就是创建那些聊天页面。

6.3.3　聊天页面模板

对于实际的聊天页面，要更巧妙些。直到现在，每次要修改任何功能，都得创建一个新的请求处理程序。但是，所有的聊天页面除了聊天室的名称不一样外，其余都是相同的——实际上，

之后我们会想动态地创建和删除聊天页面。所以，我们将通过聊天页面模板和Python代码里一些巧妙的URL处理逻辑的结合，对所有的聊天只实现一个处理程序类。Python代码会通过请求的URL识别出请求的是哪个聊天室。

基本的聊天模板和我们一直使用的老式的消息显示循环相同，但是这次，我们将聊天室的名称放在页面头部，并从每条消息中删除——毕竟，所有消息都属于同一个聊天室，没有理由需要重复保留这些信息。如下是该模板的实现：

multichat/multichat.html

```
{% extends "master.html" %}

{% block pagecontent %}

{% for m in msg_list %}
 <p> <b>({{ m.chat }})</b> {{ m.user }} ( {{ m.timestamp }} ):
       {{ m.message|escape }} </p>
{% endfor %}

    <form action="/talk?chat={{ chat }}" method="post">
      <p><b>Message</b></p>
      <div><textarea name="message" rows="5" cols="60"></textarea></div>
      <div><input type="submit" value="Send ChatMessage"/></div>
    </form>

{% endblock %}
```

除了给页面底部的登录表所生成的POST请求增加了一个参数之外，其余的和我们之前做的基本相同，之后，我们将使用该参数来传递发布消息的聊天室的名称。

该代码体现了一些巧妙之处。我们必须处理两件事情：需要验证请求，而且，需要基于请求中使用的URL来定制输出。

验证步骤是新的。直到现在，我们一直都依赖于Google App Engine来处理验证，将每个特定的URL映射到一个特定的请求处理程序上，从而非法请求会产生错误。但是现在，我们要将多个请求映射到一个单一的处理程序上：每个查看聊天室的请求都会被相同的RequestHandler处理。因为我们最终会实现增加和删除聊天室的功能，所以，没有固定的聊天室列表，能被我们用来进行app.yaml文件中或者WSGIApplication实例中的硬编码。当我们收到查看特定聊天室的请求，或者在特定聊天室发布消息的请求时，需要检查请求中的聊天室是否有效。

multichat/tchat.py

```
class GenericChatPage(webapp.RequestHandler):
  def get(self):
❶   requested_chat = self.request.get("chat", default_value=None)
❷   if requested_chat == None or requested_chat not in CHATS:
      template_params = {
          'title': "Error! Requested chat not found!",
          'chatname': requested_chat,
          'chats': CHATS
          }
```

```
            error_template = os.path.join(os.path.dirname(__file__), 'error.html')
            page = template.render(error_template, template_params)
            self.response.out.write(page)
        else:
❸           messages = db.GqlQuery("SELECT * from ChatMessage WHERE chat = :1 "
                                   "ORDER BY timestamp", requested_chat)
            template_params = {
                'title': "MarkCC's AppEngine Chat Room",
                'msg_list': messages,
                'chat': requested_chat,
                'chats': CHATS
                }
            path = os.path.join(os.path.dirname(__ file__), 'multichat.html')
            page = template.render(path, template_params)
            self.response.out.write(page)
```

❶ 我们通过解析URL得到用户请求的聊天室名称。新的URL格式包含三部分：主机名称、资源标识符以及查询。查询由一组名称/值的二元组组成。在我们的例子中，查询的二元组如下：名为chat的查询参数和请求的聊天室名称。因此，例如，要查看聊天室`Random`，URL会是http://markcc-chatroom-one.appspot.com/enterchat?chat=Random。检查HTTP请求的元素是Google App Engine程序中一项常见的工作，因此webapp提供了一组扩展的方法库，用于检查和管理URL。要从请求的URL中得到查询参数，只需使用get方法，该方法可以得到查询参数的名称，以及如果查询没有包含该参数时所返回的可选的默认值。

❷ 正如之前讨论的，我们需要检查用户输入以确保URL所请求的聊天室是有效的。我们通过将其与全局已知的聊天室列表进行对照来实现这一功能。如果请求的聊天室不存在，将显示一个错误页面。它由一个主模板派生出来的子模板生成，因此，该页面看起来像是我们应用程序生成的错误，而不是典型的“页面无法找到”错误。

❸ 如果聊天室已经存在，我们就按照一直使用的方法来显示该聊天室。和之前的处理方式唯一不同的是，在查询中，我们只选择`chat`字段和用户选择的聊天室匹配的信息。

我们已经有一个显示消息和一个发布消息的通用处理程序。还有什么没实现呢？我们还需要更新发布消息的处理程序，以从消息中获得聊天记录。这很容易：只需要在`POST`处理程序中增加一行`chat = self.request.get("chat")`，然后在`ChatMessage`的构造函数的调用参数中增加`chat=chat`。

还剩最后一部分。虽然已经有了所有请求的请求处理程序，以及需要显示的所有页面的模板，我们还需要修改应用程序代码，将收到的请求映射到适当的请求处理程序上。要实现该功能，只需要修改应用程序的`WSGIApplication`记录。现在`WSGIApplication`记录需要登录页面、通用聊天页面以及发布页面的全部信息。更新过的代码如下所示：

multichat/tchat.py

```
chatapp = webapp.WSGIApplication([('/', ChatRoomLandingPage),
                                  ('/talk', ChatRoomPoster),
                                  ('/enterchat', GenericChatPage)])
```

在本章中，我们向聊天应用程序的功能化方面迈进了一大步。我们使用模板将程序逻辑与用户界面分隔开，而且开始使用通用页面结构，定义和使用了主模版和模板扩展，从而为应用程序的所有页面提供了一个共同的外观。

不幸的是，尽管增加了这些功能，但是我们的应用程序仍然很简陋。实际上，它看上去并不像一个应用程序，而更像一堆网页。页面布局疏松平淡。在下一章中，我们将了解层叠样式表，使用它我们可以描述：在页面中如何定义和布局出一个真正漂亮的应用程序，以及如何使用模板和层叠样式表将聊天应用程序从一个简陋但功能齐全的系统转变成一个更完美的系统。

6.4　参考文献和资源

- Django模板语言：面向模板作者，http://docs.djangoproject.com/en/dev/topics/templates/。
  官方的Django模板文档。
- Google App Engine模板文档，http://code.google.com/appengine/docs/python/gettingstarted/templates.html。
  该Google文档描述了如何在App Engine应用程序中使用Django模板。
- Django 电子书 2.0，http://www.djangobook.com/en/2.0/。
  一本关于完整的Django架构的在线电子书，包括了模板语言的详细演示。

6

第 7 章

增强用户界面的美观性：模板和CSS

在上一章中，我们做了很多事情，以填补应用所缺少的功能。但是，坦率地讲，我们的应用程序仍然有些简陋（或至少也是平淡无奇的）：

Welcome to MarkCC's AppEngine Chat Room

(Current time is 2009-07-02 01:43:13.554471)

MarkCC (2009-07-02 01:42:49.129927): Hello, is there anybody out there?

Prag (2009-07-02 01:42:56.823207): Yup, I'm here.

MarkCC (2009-07-02 01:43:13.468926): Good. I'm working on chapter 3, and testing the chat code.

Name:

Message

Send ChatMessage

事实上，它看上去比一个普通的网页要糟糕得多，因为，如今几乎每一个网页都多少会用上些样式。我们需要做些工作，使其看起来好一点，不再像一个网页，而是像一个应用程序。

在这一章中，我们要将我们的应用程序组装起来，结合使用模板和CSS来给应用程序设定格式和样式。

7.1 CSS 简介

我们需要立足于设计网络浏览器的人们的工作之上。我们要使程序看起来更好，却又没有一项内容是App Engine所特有的，甚至都不是云编程所特有的。我们将要使用的是标准的、基于HTML的格式化技术。在云应用程序中，最大的不同就是：我们使用它们显示应用程序的用户界面，而不仅仅是呈现一个漂亮的网页。技术是相同的，目标却是不同的。但是，它也有难以使用

的一面，因为HTML和CSS原本不是为构建用户界面而设计的。我们将要依赖的功能实际上也是从早期云应用程序构建者那里借鉴来的，虽然这些技术能够用来完成工作，但是需要一些时间来适应。

要理解该技术，开发者需要了解它的起源。在网络初期，人们习惯于调整HTML，从而尝试生成用户界面。通过创建精心设计的、嵌套在框架中的嵌套表的结构化文档，开发者可以创建出看起来还不错的页面——前提是用户在正确的操作系统上使用了正确的浏览器版本，以及合适的屏幕大小。但是所生成的HTML的复杂性难以想象，而且非常难以维护，甚至不能在不同的浏览器之间移植。

这样的策略显然是不可管理的。把页面的内容和其应该显示的方式两者混在一起，只会使事情变得一团糟。因此，我们需要采用一种方式将页面的外观与页面的内容相分离，其解决方案就是采用被称为CSS的技术。

CSS使得网络开发者将结构与外观分离开来。HTML用来描述页面的结构——开发者基于页面的结构元素来标记页面：章节、段落和列表。CSS则完全是关于外观的，它提供了一种获取HTML文档结构元素，以及描述这些元素应该如何显示的方式。

CSS将样式与结构分离。其背后的思想与我们在之前的章节中将用户界面和应用程序的逻辑分离基本类似，CSS可将文档的结构与其外观分离。结构用HTML编写，外观用CSS描述。这样的分离方式有以下三个优点。

- **关注点分离**（Separation of Concerns）

  关注点分离是通用软件工程原则的一个特有术语：将不同的思想融合在一段代码中总会使事情变得复杂。正如前文所述，试图将HTML文档的结构与其显示方式组合在一起，就导致了不可移植的、难以维护的混乱局面。我们将会不断提到关注点分离这一原则，比如上一章中将应用程序的逻辑与页面显示分离，本章中将结构从外观分离，以及下一章中将基本的用户界面显示与数据显示分离。

- **可重用性**（Reusability）

  样式通常不是针对一个单一网络页面的。网站上的所有页面，或者组成应用程序的所有视图，通常都采用同样的样式。将样式信息分离出来，可以使开发者仅编写一次样式，然后就可以在所有的页面中重复使用，而不是一遍又一遍地产生相同样板的样式信息。每个新页面只需要包含一行声明，用来指定描述其样式的CSS文档即可。

- **灵活性**（Flexibility）

  用户可能想要更改页面样式的属性。例如，有视觉障碍的用户可能想切换到更大的、更易于阅读的字体，或增加不同元素之间的色彩对比度。如果样式表独立于文档存放，用户就更容易通知他们的浏览器使用自己特定的样式进行替换。

7.2 使用 CSS 为文本添加样式

CSS基于一个非常简单的想法：开发者声明一个结构元素，称为选择器（Selector），然后针

对选择器，再声明一系列特性/值的二元组。例如，我们想使聊天室的背景变为蓝色，并在标题文字下加下划线，就可以使用下面的CSS：

css-chat/snippets.css

```
body {
        background-color: #8888FF;
}

h1 {
        text-decoration: underline;
}
```

此代码包括两个CSS语句。由于是第一次使用选择器`body`，所以它会应用于整个HTML文件的全部内容。在样式内部，它声明了一个特性（`background-color`），并使用一个表示中等蓝色的十六进制RGB码定义该颜色值。

第二条语句则更为具体。它的选择器是`h1`，所以它仅应用于最上层的标题。它指定`text-decoration`特性，用来美化文字，为文本添加阴影、下划线和删除线等。这条语句声明一级标题应该加下划线。

要将此样式应用于我们的聊天页面，需要将其保存在一个文件中，然后在HTML页面模板中添加一个片段，告诉用户的浏览器获取该CSS文件，并用它来显示页面。如果我们把CSS放在一个名为`chat-style.css`的文件中，就可以在HTML文档的头部增加一个样式表链接，将此样式应用于该页面，如下所示：

css-chat/snippets.html

```
<head>
    <title>{{ title }}</title>
    <link rel="stylesheet" media="screen"
          type="text/css" href="chat-style.css"/>
</head>
```

利用这些基本知识，我们使用样式来构建一个新的聊天室版本。在此过程中，我们将看到如何用更灵活的方式使用CSS选择器。

先从简单的开始：把整个页面的背景改为深蓝色；欢迎的标题信息用一个大的、有吸引力的字体，并将文字设为白色，背景设置为深蓝色：

css-chat/snippets.css

```
h1 {
    font-family: 16px Helvetica, sans-serif;
    color: #FFFFFF;
    background-color: #0000A0;
}
```

这一步中唯一的新内容是如何声明字体。字体有点复杂，因为不同操作系统上的不同浏览器可用的字体集是不同的。我们并不只指定一种字体，而是按照优先顺序指定一系列字体。如果第一个

字体在显示该页面的浏览器上是可用的，浏览器就会使用该字体；如果不是，它会尝试下一个，依此类推。如果我们的样式文件中列出的所有字体在浏览器中都不可用，浏览器将使用默认字体。代码实现的工作是，字体应该以16像素的大小显示，并且字体优先选择Helvetica；如果Helvetica不可用，那么浏览器会使用其默认的sans-serif字体。

我们还为<body>元素和<h1>元素声明了`background-color`特性。这是层叠的一个极其简单的使用。样式，无论是来自多个样式表，还是来自样式表内的多条语句，当有多个声明时，内容如何解析都遵循一组规则。完整的规则相当复杂，但基本的思路是这样的：最特定的CSS声明总是被优先考虑。HTML文件中的CSS优先于链接样式表；特定页面的链接样式表优先于网站的默认样式表；嵌套元素的样式特性组优先于包含该元素的相同的样式特性组。因此，`h1`元素的背景将优先于`body`元素的背景。

接下来更新导航条。导航条会存在于许多网页上，所以我们不希望它占用太多的空间，但希望它很明显。我们将其设为小的黑色文字、白色背景。要做到这一点，需要对HTML和CSS都进行修改。对于HTML，只需对导航条块进行如下更改：

css-chat/snippets.css

```
{% block navbar %}
<p id="navbar">
{% for c in chats %}
<a href="{{ c.url }}">c.name</a>
{% endfor %}
</p>
{% endblock %}
```

我们修改了导航条，使它成为一个聊天室名称的水平栏，我们还将其封装在有注解的<p>标签内。该标签中包含了`id=`属性。ID是HTML中为使用CSS而专门添加的一种机制。有了ID，我们就能够编写一个影响HTML文档中特定元素的CSS规则。由于导航条是唯一的（我们知道聊天室永远只会有一个导航条），因而可以用一个标识符来标识它。然后，我们就可以使用选择器`#navbar`来给导航条设定样式。ID附加到什么元素上并不重要，它可以是HTML文件中的任意元素。因此，我们可以修改导航条，不管是图片，还是<div>标签，或者是<span>标签，抑或是一个列表，都可以——而且根本不需要修改CSS。下面是样式化导航条的CSS：

css-chat/chat.css

```
#navbar {
        font-family: 8px Helvetica, sans-serif;
        color: #000000;
        background-color: FFFFFF;
}
```

这里的CSS非常简单。我们使用名称为`#navbar`的ID选择器，然后声明相应的特性。

现在，我们再来看看聊天室中实际聊天时的界面。在聊天过程中，每个消息使用<p>标签表示段的显示。我们希望其中一些<p>标签用来显示来自其他用户的消息，另外一些用来显示查看

该网页的用户的消息，并且，我们希望页面上所有的消息段都与其他段落显示的不同。

HTML又一次派上用场了。有一种标记页面上标签的方法，使我们能够声明一个应该以特殊方式显示的特殊XML标签的某些实例。用户可以用`class=`属性来注解任意HTML标签，然后用户可以编写选择器，将其应用于声明了特定`class=`值的任意标签，或者应用于某一特定标签（如<p>）。在这个聊天应用程序中，我们先做一个特定的选择器。CSS样式可以为无标记的<p>标签声明一个样式，对有标记的标签声明一个特定的样式。根据通常CSS的“最特殊规则”，特殊标签的样式会覆盖通用标签的样式。因此，我们为两种聊天消息编写两种样式。对于浏览页面的用户所发送的消息，我们用纯白色文本；对于其他人所发送的消息，我们将绘制一个暗黄色的背景。实现该功能的CSS如下所示：

css-chat/chat.css

```
p.sentbyme {
        color: #FFFFFF;
        font-family: 9px Helvetica, sans-serif;
}

p.sentbyother {
        color: #DDDDFF;
        font-family: 10px Helvetica, sans-serif;
        background-color: #000080;
}
```

正如你所看到的，属性`class=`设置为`sentbyme`的<p>标签的选择器被改写为`p.sentbyme`。

要使用此CSS，需要修改HTML页面模板或Python应用程序逻辑，以使查看页面的用户所发送的消息和其他用户所发送的消息有不同的分类。请思考一下什么被改变了，然后，我们再来决定修改哪里。这里是应用程序逻辑的改变，还是仅仅是应用程序外观的变化？

在这个例子中，只是外观进行了改变，因此通过模板来实现这一功能。我们将使用模板的扩展来显示聊天页面，该扩展包括一个条件测试，以决定每条消息使用哪个类。模板如下所示：

css-chat/distinct-messages.html

```
{% extends master.html %}

{% block pagecontent %}

{% for m in msg_list %}

❶     {% ifequal msg.sender m.user %}
      <p class="sentbyme">
      {% else %}
      <p class="sentbyother">
      {% endifequal %}

        <b>({{ m.chat }})</b> {{ m.user }} ( {{ m.timestamp }} ):
           {{ m.message | escape }} </p>
```

```
{% endfor %}

  <form action="/talk&chat={{ chat }}" method="post">
    <p><b>Message</b></p>
    <div><textarea name="message" rows="5" cols="60"></textarea></div>
    <div><input type="submit" value="Send ChatMessage"/></div>
  </form>

{% endblock %}
```

在❶处，我们使用一个新的Django模板代码。ifequal标签需要两个变量，根据Python检查它们是否相等。如果相等，它输出ifequal和else之间的HTML文本；否则，它会输出else和ifequal块结尾之间的文本。因此，这段代码用来检查消息的发送者与查看该网页的用户的名称是否相同。如果两者是相同的，它将生成一个声明为sentbyme类的<p>标签；否则，它会生成一个声明为sentbyother类的<p>标签。

在CSS中经常可以省略标签的名称。我们可以声明一个样式规则，将其应用于具有特定类的任意标签。如果在上面的样式规则中省略p，那么，选择器将只是.sentbyme和.sentbyother，这些规则就可以应用于具有该类的所有标签。这是一个非常有用的技术，原因如下所述。

(1) 有一些特性可能会用在文档的很多地方。例如，对于页面中描述错误的部分，我们通常希望能用鲜红色来显示这些部分。因此可以创建一个error类。然后在输出中任何有错误文字的地方——无论它是一个头、一个段落还是小部分的黑体文本，我们都可以给其标签添加class="error"。

(2) 可以为应用程序尝试不同的布局。例如，导航栏开始时是项目符号列表，之后把它改成横向列表。我们可以使用一个类设置用户界面元素的装饰特性（如颜色和字体风格），然后当改变在HTML中编写界面元素的方式时，就不需要修改CSS了。

7.3　使用 CSS 的页面布局

我们已经明白了CSS是如何通过控制字体、颜色和装饰使界面看起来更具有视觉吸引力的。但对于构建用户界面而言，仍然缺了一些非常重要的内容——布局。为了使用户界面既具吸引力又实用，需要控制内容在屏幕上呈现的位置。将用户界面布局交给用户浏览器的布局引擎来处理不是一个明智的选择。浏览器显示简单的HTML，以使网页更易于阅读，但难以自动生成精美的用户界面。

我们的用户界面由一组区域组成。我们希望它的模型看起来如图7-1所示。用户界面有一个欢迎的标题信息，这是一个全屏幕宽的区域；一个垂直运行于屏幕左侧的导航栏；一个显示聊天记录的区域；以及在聊天记录下边的输入表单。这些元素中的每一个都基本上是一个矩形区域，我们希望它们以一种特定的方式分布。

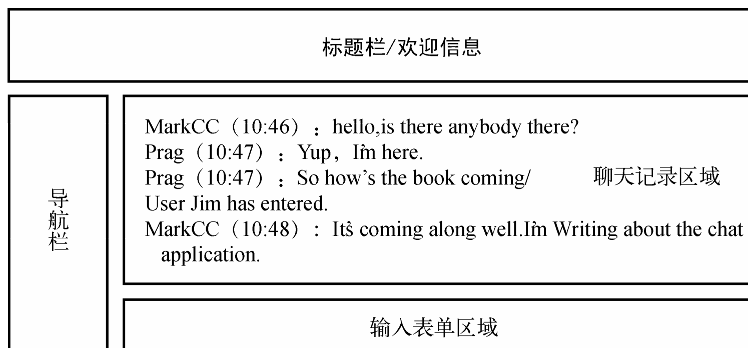


图7-1　聊天用户界面的结构

我们必须定义这些区域是什么，以及它们应该如何布局。从我们到目前为止所做的工作中，读者大概可以猜出来下面要讲的内容：有一个用于描述各区域里是什么（结构）的HTML元素，以及描述其如何布局的CSS特性（外观）。

7.3.1　用div元素描述文档结构

HTML语言的<div>元素可以包含任意其他HTML标签和元素的集合，包括其他<div>。其唯一目的就是描述由其他元素集合所组成的结构。如果没有CSS改变布局和外观，用户甚至无法看到<div>在文档中的什么地方——默认情况下，它们没有可见的特性。它们只是生成一个区域，然后用户可以通过CSS选择器引用该区域。

让我们更新一下基本页面模板，使用<div>元素来描述文档结构：

css-chat/master.html

```
<html>
 <head>
  <title>{{ title }}</title>
  <link rel="stylesheet" media="screen"
        type="text/css"
                href="chat-style-layout.css"/>
 </head>

 <body>

<div id="header-block">
    <h1>Welcome to {{ title }} </h1>
    <p> Current time is {% now "F j Y H:i" %}</p>
</div>

<div id="navbar-block">
  {% block navbar %}
    <div id="navbar">
      <ul>
         {% for c in chats %}
```

```
        <li><a href="{{ c.url }}">c.name</a></li>   {% end for %}
      </ul>
    </div>
    {% endblock %}
</div>

<div id="content">
  {% block pagecontent %}

    <p> This is template text. If you're seeing this in a page rendered
      by chat, something is wrong.</p>

    {% endblock %}
  </body>
</div>

<div id="entry-form">
  {% block entry %}
  {% endblock %}
</div>

</html>
```

这只是上一章的主模板，然后采用了图7-1中用户界面框架的每个区域，将用户界面对应部分的HTML用<div>元素围起来，并用id=属性对其进行标记。

7.3.2 基于流的布局

我们有了应用程序视图的HTML文件，视图被结构化为一组区域。现在需要编写CSS对页面进行布局。CSS提供了大量的特性，我们可以采用这些特性来管理页面布局的方式，使它看起来像一个自然的用户界面。

需要注意（总有要注意的，不是吗？），布局非常复杂。这道理再自然不过：我们可能要在任意大小的显示器及浏览器窗口内进行布局，并且还要考虑到在不同的浏览器中布局引擎还有所差异。

我们使用流约束（Flow Constraints）来描述布局。其基本思路是，如果没有CSS的布局特性，浏览器通过向页面流入文本，来实现对HTML页面元素的布局。HTML页面以代表空白行的矩形开头。文本流入该行，直到该行被填满。一旦该行被填满，就会在它下边创建一个新行，文本也随之流入新的区域。因此可以通过将文本流向屏幕上各个矩形区域，来完成页面布局：从左到右，然后从上到下。要改变布局，就得进入这个过程。例如可以将一个区域放在屏幕特定的位置，从而中断文本流入过程，于是其余的页面内容就会围绕这个区域填充。

这种在屏幕上布局的方法将基于流来实现。HTML不能只是一组按照任意原有顺序组织的<div>集合，我们将通过布局在屏幕上放置这些<div>元素——各元素出现的顺序会对它们具体的显示位置以及用户界面最终呈现方式有很大的影响。

CSS为我们提供了两个很棒的工具用于构建布局：浮动框（float）和清除（clear）。

1. 浮动框

浮动框是一个CSS样式包括了`float`属性的<div>元素（或其他HTML元素）。在布局中，浮动框（正如它的名字一样）在页面的边之间浮动。它们工作的方式很简单：按照标准的流布局进行定位，但是它并不停留在它到达的地方，而是浮动到边上，其他元素就可以围绕它流动了。

这么说还不清楚，所以，我们来看一个例子。让我们组装一个聊天用户界面的小模型，看看不使用浮动框是什么样子的，然后增加浮动框，看看它如何变化。我们将做一个导航侧边栏和聊天视图区域的模型。下面是HTML：

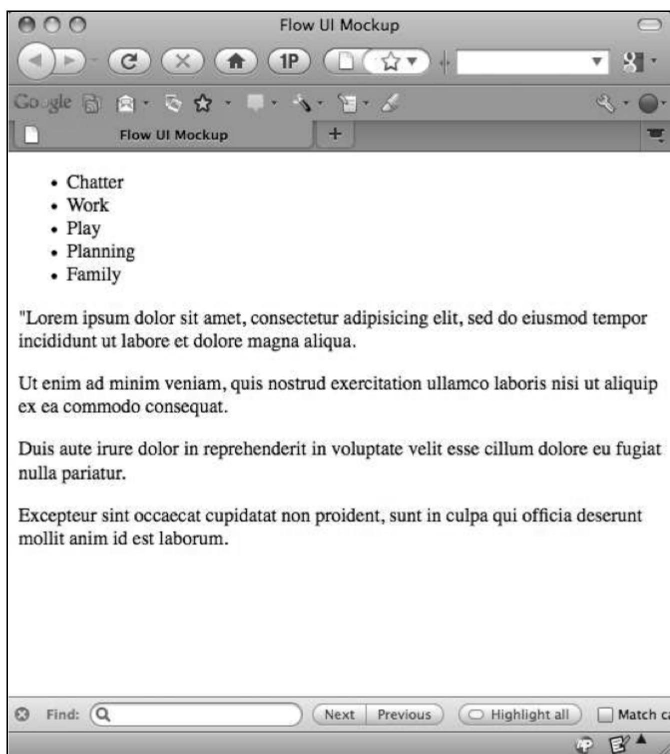
css-chat/flow-mockup.html

```
<html>
  <head>
    <title>Flow UI Mockup</title>
    <link rel="stylesheet" media="screen"
          type="text/css" href="flow-mockup.css"/>
  </head>

  <body>
    <div id="sidebar">
      <ul>
        <li>Chatter</li>
        <li>Work</li>
        <li>Play</li>
        <li>Planning</li>
        <li>Family</li>
      </ul>
    </div>

    <div id="body">
      <p>"Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do
        eiusmod tempor incididunt ut labore et dolore magna aliqua. </p>
      <p> Ut enim ad minim veniam, quis nostrud exercitation ullamco
        laboris nisi ut aliquip ex ea commodo consequat. </p>
      <p> Duis aute irure dolor in reprehenderit in voluptate
        velit esse cillum dolore eu fugiat nulla pariatur. </p>
      <p> Excepteur sint occaecat cupidatat non proident, sunt
        in culpa qui officia deserunt mollit anim id est laborum.</p>
    </div>
  </body>
</html>
```

它会生成一个页面，看起来像这样：



我们可以使用CSS让侧边栏浮动起来：

css-chat/flow-twocol.css

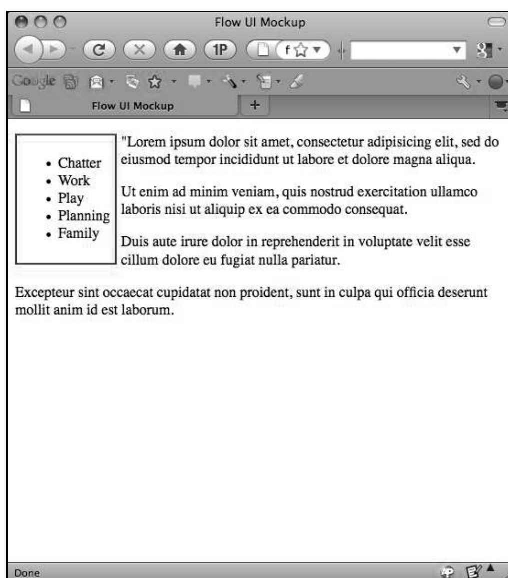
```
   #sidebar {
❶        float: left;
❷        border: 2px solid #0000FF;
❸        padding: 5px;
        margin-right: 5px;
  }
```

❶ `float`的特性可以设置为`left`（左）、`right`（右），或者`none`（没有）。`left`在我们的例子中表示该元素应向左浮动，`right`表示该元素应向右浮动，`none`可以使开发者取消继承了该类`float`特性的元素的浮动。

❷ `border`特性使开发者在该<div>的边缘绘制一个边框。它的格式为“宽度 style 颜色”。`style`描述了边界轮廓应该是什么样子，可以是`solid`（实线）、`dotted`（点线）、`dashed`（虚线）、`double`（双线）、`grooved`（凹的）、`ridge`（凸的）、`inset`（内嵌）或`outset`（外镶）。在这个例子中，我们使用简单的实线轮廓。

❸ 只需增加少量的工作就可以使界面变得更美观。CSS允许开发者使用<div>元素周围的两种空间：边距和填充。边距是<div>区域之外增加的空间，填充是区域内增加的空间。

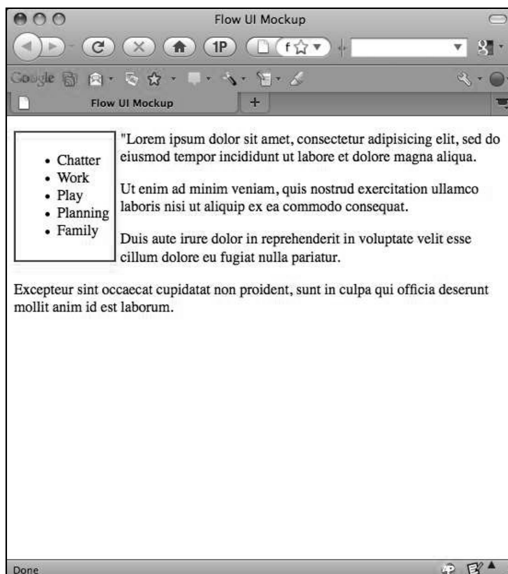
使用该样式的结果如下：



2. 清除

正如上述截图所示，主体块的文本在侧栏周围流动。它紧邻侧栏显示，一直到侧栏结束的地方，然后再流动到左边界。

我们也可以通过将聊天记录文本放在浮动框里，来阻止聊天记录文本围绕侧栏流动。如果我们只天真地设置`float: right;`，得到的界面将看起来像这样：



在默认情况下，浮动框是独立的，而对它们进行定位时，每个浮动框都认为它自己是整个页面的宽度。由于聊天记录是向右浮动的浮动框，而导航条是一个向左浮动的浮动框，在流动布局里，它们都被认为是全页面宽度的浮动框，不会并排出现，在布局里会垂直堆叠。为了让它们的位置合适，我们需要做的是指定它们的宽度。由于是在网络浏览器内部进行操作，页面宽度是可变的——所以，通常使用百分比描述位置和宽度。我们可以使用绝对测量值，但多数情况下最好使用相对值。为了按照期望方式在模型中放置这些元素，我们需要设置宽度。下面是更新后的样式表：

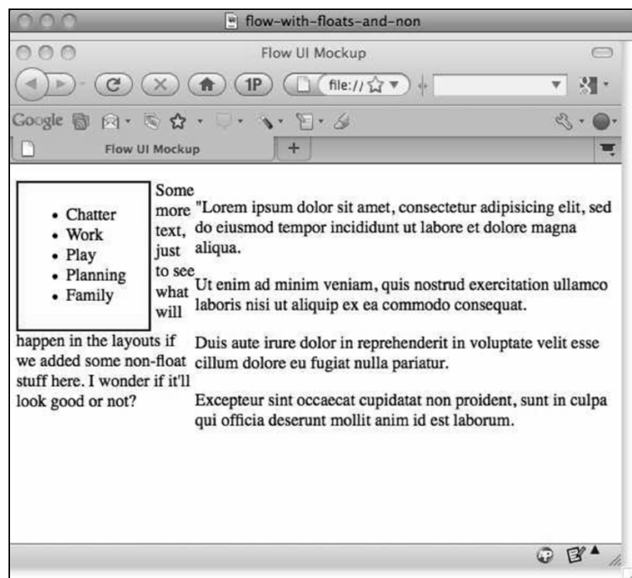
css-chat/flow-twocol.css

```
#sidebar {
        float: left;
        border: 2px solid #0000FF;
        padding: 5px;
        margin-right: 5px;
        width: 20%;
}

#body {
        float: right;
        width: 70%;
}

p.allclear {
        clear: both;
}
```

这是相应的页面：

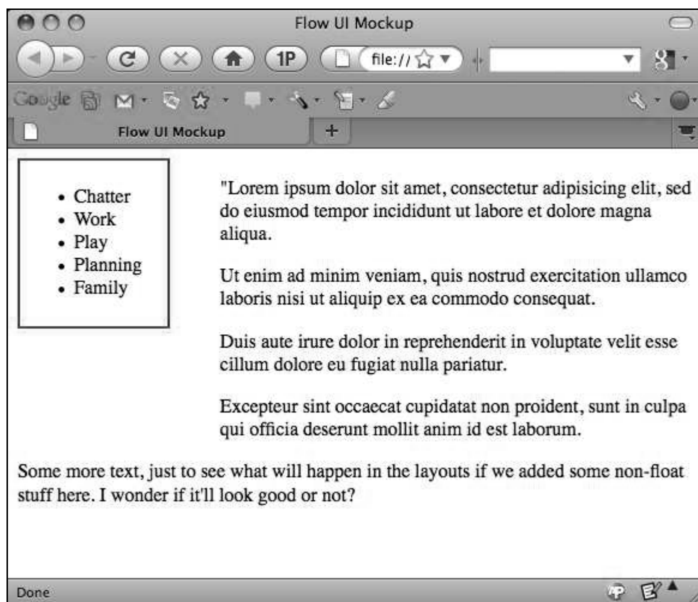


7

这样看起来还不错，但如果我们试图给用户界面添加更多内容，例如在底部的输入框（如模型所示），它会停止围绕两个浮动框流动。如果导航条比聊天记录更短，其他内容会围绕它流动到左侧，就像原始模型中那样。

我们需要的是一个“不要围绕它流动”的方式。这就是清除的用处。如果在CSS中给所有元素增加`clear: both;`，就能看到我们想要的效果。`clear: left;`清除左浮动框，`right;`清除右浮动框；或者`both;`清除所有浮动框。

给其余文本的<p>标签增加一个`allclear`类，并且给样式表增加`p.allclear{ clear: both;}`。布局结果显示在这里，正是我们所希望的：



7.4　使用流布局构建我们的界面

现在我们有了一个如何将界面组合起来的基本思路。只需再修饰一下，然后编写CSS，就可以使它成为我们想要的布局方式。一旦了解了CSS是如何工作的，实现一个看起来让人赞美的界面是很容易的——而且在此基础上得到那种很酷的界面也并不难，只是需要花点时间。我们要花费一些时间来调整各种选项、移动边界、改变颜色、重排元素，等等。这些工作需要时间，但是是值得的：看起来很酷的应用（如Gmail）和看起来一般的应用（如我们当前的应用程序版本）的区别，只是在修饰CSS上花费的时间是不同的。

回到聊天应用程序，我们的界面有四个基本元素。在应用程序的顶部有一个欢迎条；在其下方，有一个左侧的导航栏；在导航栏旁边，有聊天记录；这些内容的最下面，是一个包含新消息输入表单的长方块，它占据了窗口的整个宽度。下面是CSS代码：

css-chat/app.css

```
body {
    background-color: #8888FF;
}

#header-block {
    font-family: 16px Helvetica, sans-serif;
    color: #FFFFFF;
    background-color: #0000A0;
    border: 2px ridge #0000F0;
}

#navbar-block {
    float: left;
    width: 20%;
    font-family: 8px Helvetica, sans-serif;
    color: #000000;
    background-color: FFFFFF;
    border: 2px ridge #0000F0;
    padding: 4px;
    margin-right: 4px;
}

#transcript-block {
    padding: 4px;
    float: right;
    width: 75%;
    font-family: 8px Helvetica, sans-serif;
    background-color: 444444;
    color: #FFFFFF;
    border: 2px ridge #0000F0;
}

#entry-block {
    clear: both;
    border: 2px ridge #0000F0;
    margin-top: 4px;
}

p.sentbyme {
    color: #FFFFFF;
    font-family: 9px Helvetica, sans-serif;
}

p.sentbyother {
    color: #DDDDFF;
    font-family: 10px Helvetica, sans-serif;
    background-color: #000080;
}
```

显示结果如图7-2所示。一旦仿造界面看起来令我们满意，我们就可以在主模板中增加一个样式表链接行，将其挂接到真实的应用程序上，并上传到App Engine。

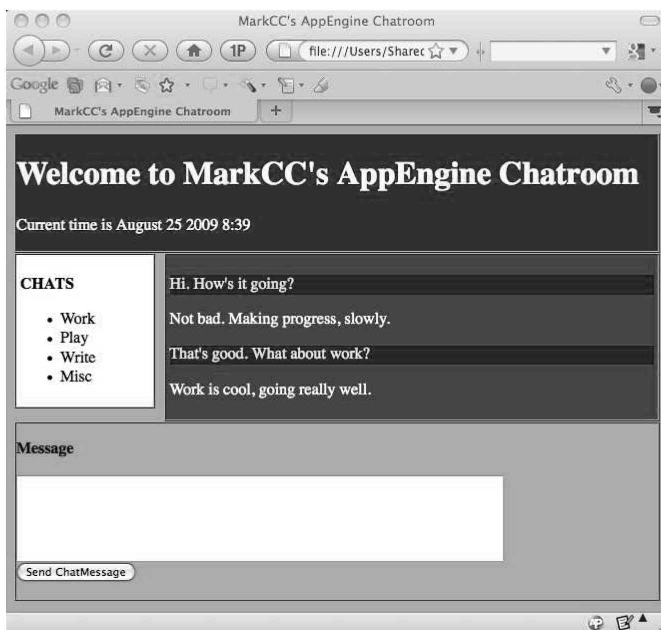


图7-2　用来测试CSS的仿造的聊天界面

这里没有什么复杂的东西，只是把前面讨论的内容组合在一起。我们使用了颜色、填充、边框、浮动框和清除来创建想要的用户界面结构。我们可以使用一个伪造的文件来测试一下这个CSS——这个伪造的文件也是一个简单的HTML文件，它所遵循的结构与我们的应用程序会生成的页面相同。该文件包含以下元素：

css-chat/fake-ui.html

```
<html>
  <head>
    <title>MarkCC's AppEngine Chatroom</title>
    <link rel="stylesheet" media="screen" type="text/css" href="app.css"/>
  </head>

  <body>
    <div id="header-block">
      <h1>Welcome to MarkCC's AppEngine Chatroom </h1>
      <p> Current time is March 13, 2011 8:39</p>
    </div>

    <div id="navbar-block">
      <p><b>CHATS</b></p>
      <ul>
        <li>Work</li>
        <li>Play</li>
        <li>Write</li>
        <li>Misc</li>
```

```
      </ul>
    </div>

    <div id="transcript-block">
      <p class="sentbyother">Hi. How's it going?</p>
      <p class="sentbyme">Not bad. Making progress, slowly.</p>
      <p class="sentbyother">That's good. What about work?</p>
      <p class="sentbyme">Work is cool, going really well.</p>
    </div>

    <div id="entry-block">
      <form action="/talk&chat=thischat" method="post">
        <p><b>Message</b></p>
        <div><textarea name="message" rows="5" cols="60"></textarea></div>
        <div><input type="submit" value="Send ChatMessage"/></div>
      </form>
    </div>
  </body>
</html>
```

7.5 在 App Engine 应用程序中包含 CSS 文件

在App Engine的Python代码中使用CSS有一些技巧。App Engine对可执行代码（动态内容）和数据（静态内容）是加以区别的。CSS不是可执行的，在App Engine的云应用程序的上下文中，它只是一个静态的数据文件。因此，开发者不能用引用模板的方式引用CSS。开发者的应用程序将生成一个HTML页面，其中包含一个引用CSS的<link>标签，但是接下来，为CSS发送请求则是客户端浏览器的责任。

因为这是一个单独的HTTP请求，我们需要告诉App Engine如何处理该请求。针对程序处理的请求，我们在`app.yaml`文件中写入`url:`项。在CSS中，我们要做的事情完全相同。我喜欢把CSS文件放到它们自己的目录中，所以对我来说，一个典型的`app.yaml`文件看起来应该像这样：

interactive/app.yaml

```
application: markcc-chatroom-one
version: inter
runtime: python
api_version: 1

handlers:
- url: /(.*\.css)
  static_files: static/\1
  upload: static/(.*\.css)

- url: /(.*\.js)
  static_files: js/\1
  upload: js/(.*\.js)

- url: /.*
  script: chat.py
```

这个文件有一个新条目，该条目处理所有以`.css`结尾的请求。任何`x.css`的请求都将被重新映射到`static/x.css`上。下一章将介绍这种运作方式。我们只是在应用程序中写了一个`app.css`的CSS链接，然后把实际的CSS文件放在`static/app.css`中。

正如本章前面展示的，给App Engine应用程序设定样式，使它看起来更美观，是一项艰巨的任务。我们需要与浏览器布局算法交互，使元素按照期望的方式放置，但是使用HTML和CSS提供的功能，已经可以达到使我们的聊天应用程序看起来不错的目标了。

最近的几章一直贯穿了关注点分离的过程：我们已经将存储与应用程序状态分离开来，将应用程序逻辑与显示分离开来，现在又将页面结构和外观分离开来。在下一章中，我们将继续这一过程，使用名为AJAX的网络技术，将用户界面控制与程序的其他方面分离，并且，这个过程会让我们的应用程序的行为看上去更像一个传统的桌面应用程序。用户将不再需要做像点击刷新按钮这样的事情去查看新的聊天消息——我们的界面控制层会自动处理这些事情。

7.6　参考文献和资源

- 《CSS：权威指南》，http://oreilly.com/catalog/9781565926226。
  一本优秀的书籍，提供了CSS的详细描述、选择器、样式属性，以及所有开发者需要的其他内容。
- 《CSS的艺术和科学》，http://pragprog.com/titles/stp-ascss/the-art-science-of-css。
  一本教科书，为如何使用CSS生成赏心悦目的网络应用程序提供了很好的指南。
- CSS教程，http://www.w3schools.com/css/。
  关于各种CSS样式和属性的交互式测试的在线CSS教程。
- 布局荟萃，http://blog.html.it/layoutgala/。
  在线资源，包含30种不同网页元素布局的HTML和CSS模板。

第8章 进行交互

在这一章中，我们将在应用程序的执行方式上有一个巨大的飞跃。我们会把聊天服务变为交互式的，而不是依靠用户不断地点击“刷新”（Refresh）才能查看新消息。为了做到这一点，我们将了解以下内容。

- JavaScript，嵌入在网络浏览器中的编程语言。开发者可以在应用程序页面中嵌入JavaScript程序，在界面中创建交互式元素。
- 文档对象模型（DOM），它将HTML表示为一个可编程的对象，让开发者可以修改使用网络页面内容的用户界面部分，而网络页面内容则可以通过JavaScript程序修改。
- AJAX（异步JavaScript和XML），一种让用户向服务器发送命令和请求，而无须重新加载页面的技术，使用户不需要重新加载页面就能处理各种操作并在界面中更新。
- 模型-视图-控制器（MVC：Model-View-Controller）范式，一种能将交互式应用程序各组成部分分离开来的强大的标准架构。

8.1 交互式网络服务：基础知识

到目前为止，我们的聊天室完全是静态的。我们在浏览器中访问一个聊天室时，不管是发布新消息，还是在浏览器中点击刷新按钮，只能看到用户上次手工加载页面时所发布的消息。这并不是我们真正期望的应用程序的工作方式。我们希望应用程序是动态的，随着底层数据的更新而不断地更新自己。在聊天室中，我们希望一旦有其他用户发布消息，就能够尽快看到他们的聊天内容，我们不希望必须手动询问应用程序是否有任何新消息发布。

问题是我们没有任何使程序交互的方式。我们使用客户/服务器的请求-响应模型建立了应用程序，事实上根本没有真正建立一个客户端。我们实现了一个服务器，并且使用了作为客户端的网络浏览器的预先打包功能。在当前版本的云编程模型中，浏览器是客户端，而它的全部工作就是显示内容。如果想要使云服务像真正的应用程序那样工作，就需要增加运行在客户端的代码。构建交互式用户界面的关键技术是AJAX，该技术可以让我们使用JavaScript代码在浏览器内运行，并且提供了丰富的客户端功能。

AJAX是如何工作的呢？首先，现代的浏览器都包括一个JavaScript解释器，JavaScript是一种可以嵌入HTML中的编程语言。使用它，开发者就可以在响应用户操作的应用程序页面中嵌入程

序。因此，当用户点击按钮时，开发者可以在浏览器中立即处理该操作。

其次，JavaScript程序可以将整个HTML页面或页面的任何部分作为一个对象管理，称为DOM（文档对象模型，Document Object Model）对象。当开发者更改页面的DOM对象时，会立即更改用户在其浏览器中看到的内容。通过应用程序页面的JavaScript和DOM对象，开发者可以创建立即响应用户操作的应用程序。

理论讲得足够了，我们来尝试一些编程。我们还不准备做得太复杂。在上一章中，为了区分不同消息，我们修改了聊天消息的显示方式。现在要做的是添加一个按钮，用该按钮的开关来实现该功能。当用户第一次加载应用程序时，所有聊天记录的消息以相同的字体显示。用户按下该按钮时，显示就会改变，使其他用户发送的消息突出显示。

我们要用CSS类处理这个步骤。于是在其他用户发送的消息上放一个CSS类，以区别于查看该网页的用户所发送的消息。当用户点击该颜色变化按钮时，会遍历DOM的聊天消息，并改变这些消息的属性，用不同的方式显示这些消息。我们给消息赋予三种不同的CSS类：`self`、`other`和`other-colored`。当用户第一次加载页面时，所有的聊天消息都用`self`或`other`标记。当用户点击“突出显示”（Highlight）按钮时，程序将扫描所有消息，把所有`other`类的消息改为`other-colored`。为了能够识别需要改变的消息，我们将其用`name=`属性标记。然后，就可以查找具有该属性的元素，修改它们的`class=`属性，从而改变消息的样式。

在把各部分内容组合到一起之前，我们先从JavaScript开始，JavaScript查找页面的聊天部分，然后遍历其结构，查找有`other`类的<p>标签。为了能够容易地查找到对应的部分，我们给聊天记录部分的<div>附加一个ID。

interactive/twiddle.js

```
❶ <script type="text/javascript">
   function highlightMessages() {
❷    var chatBlock = document.getElementById("chat-transcript");
     var chatMessages = doc.getElementsByName("other");
❸    for (c in chatMessages) {
       //将类属性改为"other-highlight"
       for (a in c.attributes) {
         if (a.name == "class") {
❹          a.value = "other-highlight";
           break;
         }
       }
     }
   }
   </script>

❺ <input type="button" value="Highlight Others"
          id="HighlightButton" onclick="highlightMessages()"/>
```

❶ 通过把JavaScript代码放在<script>标签内，将其嵌入到HTML页面中。

❷ 要更改其他用户的信息的样式，必须找到需要修改的`class=`标签元素。我们将从查找<div>元素开始。在JavaScript中，我们始终可以访问正在使用全局变量`document`显示的

文档的DOM对象。DOM提供了很多查找和管理页面元素的方法。现在，我们将使用`getElementById`，该方法返回用`id=`元素标识的值为`chat-transcript`的DOM对象，

❸ 接下来，我们要查找嵌套在聊天记录中的`name=`属性的值为`other`的消息集合，即其他用户发送的消息集合。同样地，DOM提供了一种方法，正好实现我们要做的工作——`getElementsByName`，该方法返回name属性包含特定值的元素数组。

❹ 现在，也就是最后一步，我们可以更新`class=`属性，突出显示消息了。对于要更新的消息数组中的每个元素，我们需要找到它的`class=`元素，然后更新该元素。要改变DOM中的内容，只要直接改变对象的特性就可以了，只要给其`value`特性赋一个新的值就可以更新`class=`属性。

❺ 现在我们需要做的就是将JavaScript函数挂接到用户界面上。在页面的<*form*>部分创建一个按钮元素，在按钮的`onclick=`属性中写一个JavaScript函数的调用，将该函数与按钮连接在一起。

在服务器端，我们要做的改动非常小。到现在为止，我们还没有将`name`属性应用到聊天记录的消息中。要使用`name`属性，只需要对模板修改一行代码。

8.2 模型–视图–控制器设计模式

我们开始创建聊天应用程序时的代码很简单：用几个Python编写的消息处理程序，生成显示界面的HTML代码。随着聊天应用程序的扩展，为使其更强大、灵活且具有吸引力，我们采用了新技术和新语言，以控制程序各种琐碎功能的复杂性。但是现在，这些琐碎功能的数量已经到了让人感到混乱的地步。

当前，我们的应用程序由如下内容组成：

- 服务器端用Python编写的消息处理程序；
- 使用Django编写的HTML页面模板；
- 使用JavaScript编写的用户界面交互代码；
- 使用CSS编写的用户界面布局管理。

到目前为止，因为代码始终遵循一个基本结构，所以，所有这些都是可控的。在构建整个程序的过程中，我们主要考虑的是各个独立HTML页面的代码结构。我们编写了请求处理程序，并且在每个请求处理程序中，生成了显示在用户浏览器上的完整的HTML页面。这为我们的应用程序提供了一个组织架构。

但现今我们将再进一步，使用AJAX使应用程序更具交互性。通过AJAX，我们不用再让每个页面完全由一个处理程序生成。为了使程序更易组织、可维护，必须考虑好系统的架构，需要用一种规范的方式来将各个部分联合起来，共同组成我们的云应用程序。

幸运的是，云应用程序非常适合采用一种很古老但十分强大的用户接口设计模式，即“模型–视图–控制器”模式。

假如回到图形用户界面发展的初期，你会兴奋地学习Smalltalk，它是现代图形用户界面思想

的起源，窗口、按钮、鼠标和菜单都来源于Smalltalk。在Smalltalk中，程序员使用一种三部分设计模式来构建界面。三十多年后，我们仍然在使用这种设计模式，而且这种模式可以和我们构建云应用程序用户界面的方式近乎完美地匹配。图8-1展示了一个MVC应用程序的结构。

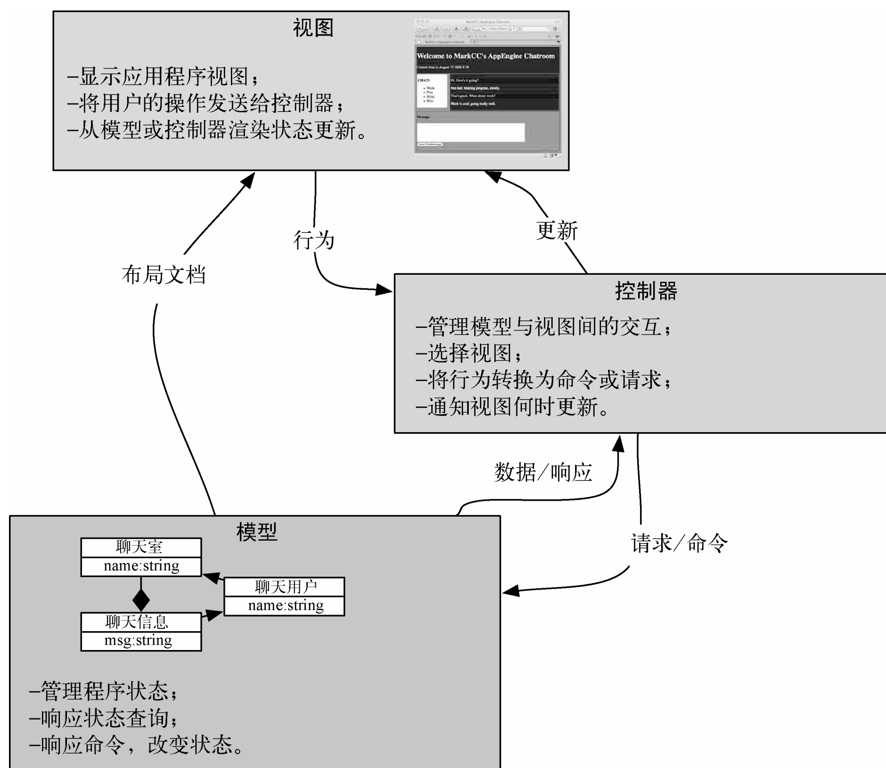


图8-1　云MVC应用程序架构

在MVC中，界面有以下3个组成部分。

- 模型（model）

  系统的应用程序逻辑。模型是围绕着应用程序计划执行的数据和操作的基本内容来加以实现的。在模型中，开发者根本不用考虑界面：模型是完全独立的，而且只代表底层应用数据工作。对于云应用程序，模型就是服务器代码。其实，服务器并不与客户端交互，它需要等待客户端发送请求。模型与用户界面完全脱离。

- 视图（view）

  视图是可见的用户界面，如显示元素、输入框，等等。在云应用程序中，视图是用HTML编写的，它没有自己真正的行为：视图只是一些显示元素，以及需要以吸引人的方式显示这些元素的代码（用HTML和CSS编写）。视图并不关心应用程序的逻辑或数据，它只要按需显示内容即可。

- **控制器**（controller）

  控制器是客户端和服务器之间的桥梁。迄今为止，我们忽略了这一关键部分。控制器获取视图中产生的界面操作，将这些操作转换为模型中可执行的操作；控制器获取服务器生成的内容数据，将此数据转换成视图可以显示的格式。在云中，控制器就是在用户的浏览器中执行的JavaScript代码。

对于我们构建云应用程序来讲，MVC是一种很自然的方式，因为MVC模型的概念和我们构建的应用程序的各个部分能够很好地匹配。我们要构建一个好的云应用程序，需要使用不同的编程语言，这迫使我们将应用程序至少分为3个部分：服务器代码（用Python或Java编写）、显示代码（用HTML及CSS编写），以及交互代码（用JavaScript编写）。使用MVC模式，我们可以更好地理解各功能与这三个部分间的对应关系。此外，使用这三种基本部分设计应用程序具有悠久的历史，我们可以在设计云应用程序时发挥其优势。当然，云计算是独特的，其体系结构与老式的Smalltalk用户界面并不完全相同，但基本思想没变。两者主要的区别是，在云MVC中，模型和视图直接交互得比较少。而在传统的MVC中，当模型中的数据被修改时，模型直接给视图发送更新。在云MVC中，一旦视图被创建了，模型就不再与其直接交互，而是需要将模型的所有更新发送给控制器，然后由控制器来更新视图。

只要思考一下就会发现，MVC正是我们一直所做的云应用程序开发过程中的下一步。由于应用程序变得更加复杂了，我们已经对一些内容进行了解耦，也就是分离了各部分。我们将显示从计算中分离，把显示用模板实现，计算用Python实现；我们将样式与内容分离，把内容放在HTML中，样式放在CSS中。现在，我们只是用相同的方式再增加一个分离层。我们的客户端既有用户界面（内容和样式），也有可执行代码，要将其分到视图（用HTML和CSS）和控制器（JavaScript）中。

往下继续开发聊天应用程序时，要牢记MVC。它是描述云应用程序被划分为逻辑块的最好的基本结构。在本章中，我们已做的工作是：将聊天程序分为模型（服务器，有存储聊天记录的数据仓库，以及管理与客户端交互的查询处理程序）、视图（提供基本用户界面的HTML和CSS文档）和控制器（进行所有交互的JavaScript）3个部分。

8.3 与服务器不中断地交互

在本章开始，我们使用JavaScript在用户界面中实现了动态交互，而不需要与服务器进行交互。这种方法很有用，但其作用也极为有限：我们只能让用户浏览器中显示的HTML文档中的数据创建各项功能。对于需要更多其他数据的工作，我们也无能为力。

当然，我们希望能够完成那些依赖从服务器获取数据的工作。虽然可以从服务器获取我们已知的数据，但必须在浏览器的客户端—服务器交互的标准同步周期内获取。我们编写客户端代码向服务器发送请求，服务器响应这些请求，并且当客户端浏览器收到响应时再将其显示出来。如果不做一些特殊的工作，就不可能避开基本的请求—响应—显示过程。

这种请求—响应—显示周期不能满足云应用程序的处理需求。我们想要编写的是，收到新聊

天消息即能自动更新显示的聊天应用程序，但聊天应用程序只能等待用户手工刷新。当然，我们也可以编写一个带计时器的JavaScript，一秒自动刷新一次。但每次发生刷新操作时，会对用户造成很大干扰。用户的界面会变为空白，然后重绘。如果他们开始输入一条新消息，那么在浏览器刷新时该消息可能会丢失，这非常糟糕。

我们需要做的是与服务器进行异步交互。也就是说，需要编写一些代码，其运行独立于通常的“请求-响应-显示”周期。这种技术被称为AJAX，即异步JavaScript和XML。AJAX使用一个称为XMLHttpRequest的JavaScript构造方法。坦白地说，XMLHttpRequest很像一个蹩脚的黑客。与我们开发基于网络的云应用程序客户端时所用的很多工具一样，它基本上是基于一些人的特需要而攒起来的一个特别的功能，而不是为构建用户界面精心设计的工具集中的一部分。当人们开始使用它，它才慢慢成了一个标准，而且现在我们非常乐意继续使用它。（我们将在9.3节中看到，AJAX可以被封装起来隐藏在幕后，这样开发者就不需要处理这些简陋的内容了，但理解一下背后的真实运作情况总还是不错的。）

XMLHttpRequest的关键在于，它是异步的，这就意味着发送一个请求时，发送者并不等待响应。通常，我们编写生成HTTP请求的代码时，使用的是函数调用：调用一个方法发送请求，而该函数调用的返回值是对查询的响应。此时，代码会阻塞——它什么都不做，只是等待，直到收到响应。但是XMLHttpRequest不会阻塞。相反，当我们创建了一个XMLHttpRequest，其参数之一称为*回调函数*（callback）。当客户端浏览器收到该请求的响应，会调用回调函数。因此，客户端永远不会阻塞——它只是继续运行，于是，用户看到的就是一个非常流畅的用户界面，没有任何中断或刷新。一旦收到响应，回调函数就会被调用，用最新的信息更新用户界面。

这个描述听起来很抽象。但是在App Engine中实现起来并不难，最好用代码来说明这个问题。我们直接开始编写真正想实现的内容——动态更新的聊天视图。

要实现聊天视图的动态更新，我们需要编写一组代码：

(1) 一个服务器请求处理程序，该程序用于提供没有任何聊天数据的用户界面框架；

(2) 一个获取聊天数据（新消息）的请求处理程序；

(3) 一个JavaScript组件，从服务器请求更新，然后在收到新数据时将消息加入聊天记录。

XMLHttpRequest 并不是 XML 的 HTTP 请求

XMLHttpRequest 的命名确实差劲。实际上，它不是一个请求，而且和 XML 没有任何关系。

XMLHttpRequest 是请求的管理者——一个用于创建请求、发送请求，以及处理其收到的响应的对象。

XMLHttpRequest 的最关键的特点在于，它能够在通常的浏览器的“请求—响应—显示”周期之外生成和发送请求，而且能够异步处理请求，因此，开发者的程序不需要等待响应。

8.3.1　模型：聊天室的请求处理程序

我们需要做的第一件事情是建立一个服务器上的请求处理程序。我们把请求处理程序分为两部分：向客户端发送用户界面的处理程序和向客户端发送数据的处理程序。前一个处理程序向客户端发送用户界面，基本上是将视图和控制器发送到浏览器，因此它可以作为我们的应用程序的客户端来运行。然后，控制器将向服务器发送数据请求，数据由第二个处理程序发送。

基本的请求处理程序是微不足道的。我们将采用的模板与一直在用的模板相同，创建它的一个特例，产生一个空白的记录区。模板将在后面的8.3节中讲述。该处理程序是我们一直使用的一个处理程序，只是使用了模板。

interactive/chat.py

```
class InterfaceServerHandler(webapp.RequestHandler):
  def get(self):
❶   requested_chat = self.request.get("chat", default_value="none")
❷   if requested_chat == "none" or requested_chat not in CHATS:
      template_params = {
          'title': "Error! Requested chat not found!",
          'chatname': requested_chat,
         'chats': CHATS
          }
      error_template = os.path.join(os.path.dirname(__file__), 'error.html')
      page = template.render(error_template, template_params)
      self.response.out.write(page)
      else:

        messages = db.GqlQuery("SELECT * from ChatMessage WHERE chat = :1 "
                             "ORDER BY timestamp", requested_chat)
        template_params = {
            'title': "MarkCC's AppEngine Chat Room",
            'msg_list': messages,
            'chat': requested_chat,
            'chats': CHATS
            }
        path = os.path.join(os.path.dirname(__file__), 'interface.html')
        page = template.render(path, template_params)
        self.response.out.write(page)
```

对于数据，我们处理起来稍有不同。到现在为止，我们已经使客户端操作处在一个无状态（stateless）的模式中——也就是说，客户端从来不基于它记录的内容向服务器发送任何消息。但是在交互式操作中，客户端运行时，会不断地向服务器发送请求，而我们希望只发送新消息——也就是说，只发送上次客户端从服务器获取数据以后的消息。因此，请求中要包括时间，而且只响应发送此时间之后发布的消息。

我们还需要在发送给客户端的响应中包括时间。毕竟，我们在云中：我们的客户端和服务器在不同的计算机上，可能在世界的不同地区。各个计算机的时钟设置可能不同，而且在服务器发送响应和客户端收到响应之间会有一个时间延迟。为了确保客户端不丢失任何消息，需要知道当

服务器在发送最后一条消息给客户端时，服务器所认定的时间。

实现所有这些功能的代码非常简单，只需要发送一个XML文档。顶层元素是我们自己的一个<ChatUpdate>元素，其中包括`time=`元素。在<ChatUpdate>内部，我们放置了HTML的<p>标签，包含聊天消息。和往常一样，我们将使用Django模板来生成XML。模板非常简单：

interactive/update.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<ChatUpdate chat="{{ chat }}" time="{{ time }}">
{% for m in msg_list %}
<p>({{ m.chat }}) {{ m.user }} ( {{ m.timestamp }} ): {{ m.message|escape }} </p>
{% endfor %}
</ChatUpdate>
```

模板中使用的请求处理程序也非常简单：

interactive/chat.py

```
   class DataRequestHandler(webapp.RequestHandler):
       def get(self):
           requested_chat = self.request.get("chat", default_value="none")
           messages = db.GqlQuery("SELECT * from ChatMessage WHERE chat = :1 "
❶                                 "ORDER BY timestamp", requested_chat).fetch(20)
           template_params = {
            'msg_list': messages,
            'chat': requested_chat,
                   'time': self.request.get("time", default_value="0"),
            }
           path = os.path.join(os.path.dirname(__file__, 'update.xml')
           page = template.render(path, template_params)
           self.response.headers["Content-Type"] = "application/xml"
           self.response.headers.add_header("Access-Control-Allow-Origin", "*")
           self.response.headers.add_header("Access-Control-Allow-Methods", "GET, POST,
                                            OPTIONS")
        self.response.out.write(page)
```

这就是模型部分。正如读者所看到的，模型在交互模式下工作，确实并不需要太多特殊内容。开发者唯一需要做的是分离界面服务过程和数据服务过程。

8.3.2　控制器：客户端的JavaScript程序

上一节的服务器讨论中说道，会在初始化处理程序中发送用户界面的JavaScript控制器代码，但在代码中，我展示的所有内容就是一行包含在JavaScript中的脚本。现在，要实现该控制器。让我们直接进入代码：

interactive/js/asynch.js

```
       function SetUpAndSendRequest(time, chat) {
❶        var request = new XMLHttpRequest();
❷        var transcript = document.getElementById("chat-transcript");
```

```
❸      request.onreadystatechange = function() {
         // 等待完成请求。请求完成的标志是 ready state 为4.
         if (request.readyState != 4) {
           return;
            }
❹        var xmlData = request.responseXML.documentElement;
         if (xmlData != null) {
              transcript.innerHTML = "";
         messages = xmlData.getElementsByTagName("p");
❺        for (var x = 0; x < messages.length; x++) {
           transcript.innerHTML += "<p>" +
               messages[x].childNodes[0].nodeValue + "</p>";
             }
❻            newtime = xmlData.getAttribute("time");
❼            SetUpAndSendRequest(newtime, chat);
           } else {
             transcript.innerHTML +=
                 "<p>Sorry, but there was an error updating this chat</p>";
           }
         }
❽        request.open("GET", "latest?time=" + time + "&chat=" + chat, true);
         request.send();
     }

     function init() {
       SetUpAndSendRequest(0, "book")
     }

     window.onload = init;
```

❶ 首先，我们设置了几个需要使用的全局变量。其中最重要的一个是XMLHttpRequest。此外，我们设置了对主应用程序URL的引用，以及对HTML页面中包含聊天记录部分的引用。

❷ 从此开始实际的请求代码。要做的第一件事是打开一个请求。XMLHttpRequest对象并不是请求，它是我们用来创建和发送请求的对象。为了让其发送请求，需要首先打开一个新的请求，告诉它请求的参数是什么。

❸ 现在我们进入AJAX异步部分的核心之处。不用等待响应，我们提供了一个函数，该函数在XMLHttpRequest对象收到响应时会被调用。在这个例子中，我们把回调函数写为内联函数。

XMLHttpRequest的主要回调函数称为onreadystatechange。当请求的状态有任何改变时，函数就会被调用。请求改变状态时，可能处于各种不同的状态，如果能够响应每种不同的状态，所构建的应用程序就会更强大，但是，这超出了本书讨论的范围。有很多AJAX方面的优秀书籍可以参考——如果开发者要做很多交互式的云开发，那么确实应该阅读这么一本书。本章结尾的资源部分列出了一些值得一读的优秀书籍。

在这里，当程序收到完整的结果时，会根据请求的结果做出一些回应。这就是所谓的ready state 4。首先，我们需要在回调函数中，确保是在状态4。如果不是，回调函数返回。该回调函数会在下一次状态变化时再次被调用。

8

❹ 当回调函数最终在`ready state 4`下被调用时，我们知道从服务器收到了最新的聊天数据，因此，就可以更新聊天视图了。我们使用`XMLHttpRequest`的`responseXML`字段从响应中抓取数据。所有的聊天消息都将被放置到<p>标签中，所以要取回该标签。

❺ 追加新消息，更新聊天记录。

❻ 至此，用户界面已经用最新的聊天消息更新了。但是，我们需要重新发起`XMLHttpRequest`，使得当更多的消息发布时，可以再次更新显示。我们不希望重新获取任何已经显示的消息，因此，我们使用了响应发送的时间。

❼ 最后，我们调用此函数发送请求，实际上，我们实现了一个循环：`SetUpAndSendRequest`通过`XMLHttpRequest`创建和发起请求，编写一个收到新数据时更新用户界面的回调函数。回调函数会再次调用`SetUpAndSendRequest`。依此类推。

这是基本的模式——我们使用JavaScript创建一个`XMLHttpRequest`，它发出请求，然后设置一个回调函数。

8.3.3　聊天视图

新的交互式MVC聊天服务的视图是最容易的部分。它只是一个模板扩展，和我们之前已经做过的工作几乎相同：

interactive/interface.html

```
{% extends "master.html" %}

{% block script %}
<script src="asynch.js" language="javascript">
</script>                                              ❶
{% endblock %}

{% block pagecontent %}
<div id="chat-transcript">                             ❷
</div>

<form action="/talk?chat={{ chat }}" method="post">
   <p><b>Message</b></p>
   <div><textarea name="message" rows="5" cols="60"></textarea></div>
   <div><input type="submit" value="Send ChatMessage"/></div>
</form>

{% endblock %}
```

❶ 我们的新模板扩展必须包含含有控制器代码的JavaScript源文件。<script>标记可以使用内联代码，或者来自外部源文件的代码。这里，我们从源文件中引用。

❷ 我们需要创建一个id为`transcript`的空<div>，JavaScript将用该<div>插入聊天消息。所包含的JavaScript，包含用来启动交互过程的代码，并且处理聊天消息的显示。

通过这些代码，就基本完成了各项工作。我们已经有了一个聊天系统，其中，客户端不断地

与服务器握手，以获取最新的聊天消息，在新聊天消息发布时更新聊天记录视图。我们需要更新app.yaml文件名，以确保该文件包含了所有的新模板文件。我们还需要更新Python应用程序对象，将请求路由到正确的处理程序。我们已经非常接近基本的聊天软件所做的工作：有了多个聊天室、用户登录、完全的交互性，以及一个漂亮的界面。无论是从代码中分离数据，还是从代码中分离模型、视图和控制器，我们都已经设立了一个坚实的体系架构，它将系统划分成了各个部分。

我们可以进行一些更复杂的工作了！Google App Engine不限制开发者必须用Python编写代码，所以，开发者也可以使用其他语言。目前，常见的另一种选择是Java。在下一章中，我们将了解如何使用Java，而不是Python，来编写我们的应用程序。然后，我们将进入一些高端主题，例如高级数据仓库、安全性和调试。

8.4 参考文献和资源

- XMLHttpRequest，http://www.w3.org/TR/XMLHttpRequest/。
  官方W3C标准文档，描述JavaScript的XMLHttpRequest。
- AJAX教程，http://www.w3schools.com/Ajax/Default.Asp。
  一个AJAX在线交互教程。
- 《Ajax：权威指南》，http://oreilly.com/catalog/9780596528386。
  一本优秀的AJAX教科书。

Part 3

第三部分

用 Java 进行 Google App Engine 编程

本部分内容

第 9 章

Google App Engine和Java

我们一直在使用Python进行App Engine服务和应用程序的开发。对于许多应用程序来讲，Python是一门很了不起的的语言。实际上，对于多数开发人员来说，Python就已经够用了。就目前为止我们一直做的工作而言，Python非常优秀：友好，并且是轻量级的，正是这种轻量级使我们能够一层一层地构建应用程序，从而更清楚、详细地了解云应用程序内部各层次到底是如何工作的。

说实话，我更喜欢在编译时就可以捕获一些愚蠢的错误，而不是等到程序出错时才在浏览器中看到一堆错误信息。在给本书编写Python代码时就遇到了一个问题，有一次，我从请求处理程序给模板传递一个字符串作为参数，期望得到一个字符串列表。然而，得到的结果却是浏览器窗口中难看的一大堆转储信息。如果错误可以提早捕获的话，就不需要在运行时处理这样的错误了。

这就涉及到了Java。App Engine主要支持两种编程语言：Python和Java。Python是轻量级的动态语言，Java是重量级的。在App Engine中，开发者使用GWT工具包构建Java应用程序，（毫不夸张地说）GWT绝对是天才之作。

即使你十分喜欢Python，也有一些很充分的理由使你想在云应用程序中使用Java，例如以下这些。

- 强类型

  强类型可以捕获多种编程错误。根据开发者的编程风格，强类型可以在编译程序时捕获很多错误，使开发过程变得更容易。这在像云这样的环境中是特别有价值的，因为在云环境中很难调试程序。开发者不可能仅使用一个调试器来探查程序，也不可能只靠添加打印语句就能找到哪里出了错。任何有助于提前发现问题的手段，都可以节省开发者大量的时间。

- 风格

  稍后在本章可以看到，使用Java开发云应用的风格和结构与在Python中完全不同。对于很多开发者而言，App Engine中的Java开发风格可能比Python更令人舒服。

- 工具

  Google发布了一套针对免费Eclipse集成开发环境的插件，用于构建Java和GWT的App Engine服务和应用程序。Eclipse是一款非常优异的工具，再加上App Engine插件，一切都变得更容易了。（采用Python的开发者也可以使用Eclipse，只是因为没有专门的App Engine支持，所以这样使用起来会相当痛苦。）

在这一章中，我们将了解如何使用GWT开发云应用程序。我们把聊天应用程序移植到Java/GWT上，从而实现本章的目标。先来快速回顾一下前面的内容，看看这次如何用Java来完成之前做的工作。

静态语言与动态语言

程序员间激烈辩论的话题之一就是静态语言与动态语言孰优孰劣。因为双方都有强有力的论点论据，所以，这将是一场永无休止的辩论。两种语言的基本区别在于检测错误的时机不同。在动态语言中，错误只在代码运行时才会捕获。例如，在一个动态语言中（如Python），如果开发者编写一个类似 `x.foo()`的方法调用，而 `x` 并没有一个“foo”方法，那么要等到该语句真正执行时，开发者才会得到一个错误信息。

在静态语言中，开发者需要声明事物的类型，然后，使用这些声明所提供的信息，就能够在编译时捕获到像上边例子中的未定义方法之类的错误。

这只是一个权衡取舍的问题。在动态语言中，开发者不需要写类型声明，也就不需要向编译器证明该程序是正确的。这样做非常方便，而且可以形成一种编程风格，使用这种风格的代码非常简洁明了，并且，简单的代码中不太可能有难以发现的错误。

但是，另一方面，静态语言可以为开发者捕获很多错误。它促使开发者采用更加严格的编码方式，从而确保程序可以通过编译，而这个过程会使开发者养成良好的代码设计习惯。

就个人而言，我倾向于选择静态语言。我发现，长期以来，类型检查系统所做的额外工作为我节省了很多精力和时间。我犯的大多数愚蠢的错误都能被编译器捕获，从不会因为这些愚蠢的错误而导致程序运行时出现问题。事实上，我个人偏好强类型语言，像函数型语言 ML。ML 的类型系统有令人难以置信的表现力和难以想象的严格性，比我们通常熟悉的静态语言（如 Java 和 C ++）要强很多。作为回报，我的 ML 程序几乎没有出现过运行时错误。我的几乎所有错误最终都反映为类型的不一致，这些错误编译器都能够捕获。我已经用 ML 写了数千行代码规模的程序，花了几天时间，消除了编译器的静态检测类型错误之后，这些程序首次运行均没有出现任何错误。

9.1 GWT 简介

使用Java会比使用所有其他语言都优越，原因就是开发者可以使用GWT。GWT非常棒，通过使用GWT，开发者可以用Java编写整个云应用程序。服务器端采用通常的方法编译Java代码，生成运行于JVM之上的Java字节码。在服务器端，GWT是一个不错的框架，但并不是很出众。不过别忘了，除了服务器端以外，GWT还适用于客户端。

GWT可以使开发者用Java编写客户端程序。用Java编写客户端，就像编写传统的GUI应用程序一样：使用布局管理器从部件集合构建用户界面，添加事件处理程序，等等——这绝对是典型的GUI代码。但GWT会将此GUI代码转换为HTML和JavaScript：GWT并不是将Java编译为Java字

节码，而是将Java编译为JavaScript源代码，然后在客户端上执行。对客户端和服务器端通信所需要的所有AJAX相关内容，GWT可以生成远程过程调用。这不是一个完全自动化的过程，但是它比手工编写AJAX代码更加容易，而且代码健壮性更好。（说实话，当我听到这个消息时，我的第一反应是“他们疯了，真荒谬！”，但这正好也说明了我为何发不了财也出不了名。）

因为GWT有它自己的设置方式，所以，用GWT构建应用程序与我们使用Python的webapp所做的工作是不同的。在接下来的第一个例子中，我们将会呈现一个漂亮的用户界面，并且不需要深入了解如何设置模板和CSS的浮动框就可以实现这个漂亮的用户界面——我们将直接切入，让GWT做它最擅长的工作。

在很多方面，用GWT编程很像使用传统桌面的GUI框架编程。开发者用几乎与传统的桌面应用相同的方式定义用户界面，而GWT负责生成应用程序运行所必需的绝大多数的HTML、CSS和JavaScript。Google近期的大多数应用程序（包括像Wave等程序）都是使用GWT实现的。

为了初步了解GWT，首先请下载针对Java的App Engine SDK。我不打算详细地讲这方面的内容，因为它与第2章的下载Python SDK的过程基本相同。除了基本框架，开发者还可以为Eclipse安装一组插件，提供一个优秀的编程环境。我强烈建议下载Eclipse和App Engine插件，能使用Eclipse插件进行App Engine开发，也是使用Java工作的极佳理由！Eclipse是免费的，而且它确实很容易安装。GWT的一个缺点是有很多元数据（metadata），这也就意味着会有更多的文件用于告诉GWT如何处理Java源代码，比如哪部分编译为客户端的JavaScript，哪部分设置为服务器端的服务包，等等。由开发者维护这些文件是很痛苦的，然而，Eclipse工具则为此提供了巨大的帮助。不使用Eclipse进行GWT编程是非常不理智的。从此处开始，我假设读者使用的是有GWT插件的Eclipse开发环境。

GWT制定了一套与众不同的构建云应用程序的方法。用Python和webapp开发时，一切都以服务器为中心。当然，我们实现了客户端用户界面，但是，在其实现过程中，我们关注的是为生成客户端上的用户界面服务器需要做些什么。也就是关注构建请求处理程序，以及请求处理程序所需要的CSS和模板。GWT几乎完全相反，在GWT中，开发者将重点放在客户端。开发者使用框架构建客户端用户界面，看起来就像传统的客户端应用程序。当所开发的客户端需要服务器的内容时，开发者进行远程过程调用（RPC，Remote Procedure Call），而GWT会处理好将RPC转换为AJAX调用的大部分工作。

请记住这一点，现在让我们开始构建一个GWT应用程序。

9.2　Java 和 GWT 入门

首先，我们来看一个基本的“Hello, World”程序。Eclipse的GWT工具自动生成一个项目框架，这就是一个基本的GWT的“Hello, World”程序。因此，不用自己编写，我们只需让Eclipse完成即可，然后分析代码段，看看这一切是如何组合在一起的。在Eclipse集成环境中，从“文件”（File）菜单选择“新建”（New）。在出现的对话框中，选择“新建Web应用程序项目”（New Web Application Project），然后在出现的对话框中，填写项目名称和Java代码中要使用的Java包的名称。

我选择HelloChat作为项目名称，选择com.pragprog.aebook.hellochat作为Java包的名称。

初始应用程序会启动一个页面，提示用户输入名字。用户输入名字后，会弹出一个对话框，显示欢迎语句。

9.2.1 GWT应用程序的结构

GWT应用程序由一组模块（module）组成。模块就是一种包，内容包含Java代码、JavaScript代码、HTML文件、图像、数据定义，以及开发者在Web应用程序中所需要的其他任何内容。开发者在Eclipse集成环境中创建一个GWT/App Engine项目时，所得到的目录结构的基础就是它所实现的GWT模块的结构。

首先来看看这个目录结构。开发者可以在Eclipse包浏览器中看到该结构，如图9-1所示。在App Engine项目里面，有GWT库的集合，外加两个主要的组成部分：一个名为src的源文件目录和一个名为war的目标文件目录。war表示“网络存档”（web archive），开发者上传到App Engine的可部署应用程序就是一个war文件。

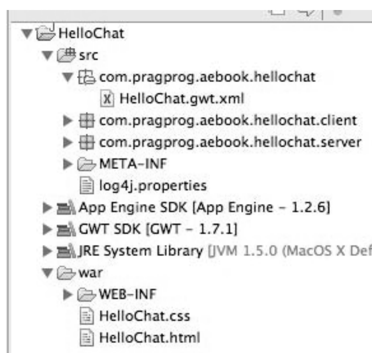


图9-1 Eclipse开发环境中的GWT项目目录结构

源文件目录本身也分为3个部分：模块声明、客户端Java代码包和服务器端Java代码包。

服务器包com.pragprog.aebook.hellochat.server看似非常简单，只有一个微不足道的源文件，因为GWT会自动生成服务器端的基础结构。

客户端有三个文件，其中，HelloChat.java是我们的应用程序的主体，另外两个是GreetingService.java和GreetingServiceImpl.Java，是GWT远程过程调用的安装部分。这些文件包含GWT所需要的声明，因此不必显式地设置许多XMLHttpRequest就可编写AJAX客户端/服务器应用程序。我们将在9.3节中学习一下这些文件是如何工作的。

这些部分组合在一起的方式是由GWT模块声明决定的。

workspace/HelloChat/src/com/pragprog/aebook/hellochat/HelloChat.gwt.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE module PUBLIC "-//Google Inc.//DTD Google Web Toolkit 1.7.1//EN"
```

```
        "http://google-web-toolkit.googlecode.com/.../gwt-module.dtd">
❶  <module rename-to='hellochat'>

      <inherits
❷         name='com.google.gwt.user.User'/>

      <inherits
❸         name='com.google.gwt.user.theme.standard.Standard'/>

      <entry-point
❹         class='com.pragprog.aebook.hellochat.client.HelloChat'/>
   </module>
```

❶ GWT中的基本代码单元是模块，模块是很多内容的集合，包括Java代码、资源（如CSS、HTML或图像文件）和GWT的定制（如Java到JavaScript编译器的扩展）。当前行声明了包含我们应用程序的模块。`rename`元素是GWT的URL处理的一部分：GWT将会告诉服务器，在以`hellochat`结尾的URL路径上安装该模块。

❷ GWT中的模块可以从其他模块继承，其工作原理很像面向对象的继承。我们的应用程序是`com.google.gwt.user.User`的子模块，这是针对有用户界面的应用程序的一个标准模块。多数GWT的基本功能——用户界面部件、远程过程调用基础架构和基本的服务器端的服务设施——都通过此标准模块的声明继承。

❸ GWT定义模块有两方面原因，一是可以使用Java代码的类继承，二是GWT模块中除了代码还有很多资源。模块可以包含很多东西，如CSS。该继承声明将定义用户界面组件外观的CSS文件放到我们的应用程序中。我们可以通过继承不同的样式模块来修改应用程序的外观。

❹ GWT应用程序的Java代码从*入口点*（entry point）开始。入口点基本上就是GWT图形用户界面的`main`函数。在模块文件中，开发者声明了想要在GWT应用程序中执行的代码入口点。在这个例子中，入口点是`HelloChat`类。

9.2.2 在GWT中设置用户界面

在GWT模块内，用户界面框架由HTML文件定义。 HTML文件不是源代码，因此，不会被放在`src`目录下。它是一个静态资源，包含了代码要使用的信息。因此，这个HTML文件被放在`war`目录下。让我们来看一下它的内容：

workspace/HelloChat/war/HelloChat.html

```
   <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
   <html>
❶     <head>
         <meta http-equiv="content-type" content="text/html; charset=UTF-8">
         <link type="text/css" rel="stylesheet" href="HelloChat.css"/>
         <title>Web Application Starter Project</title>
         <script type="text/javascript" language="javascript"
❷               src="hellochat/hellochat.nocache.js"></script>
     </head>
```

```
❸    <body>
        <h1>Hello World</h1>
        <table align="center">
          <tr>
❹          <td colspan="2" style="font-weight:bold;">Please enter your name:</td>
          </tr>

          <tr>
❺          <td id="nameFieldContainer"></td>
            <td id="sendButtonContainer"></td>
          </tr>
        </table>
      </body>
    </html>
```

❶ HTML框架文件是一个标准的HTML文件。文件开头是通常的HTML的内容：DOCTYPE声明，以及包含通常的元标签的头部。

❷ 这是整个文件中最重要的一行！之所以该HTML文件需要放入GWT应用程序框架，就是因为这个包含了JavaScript文件的代码行。该行将放入由GWT生成的JavaScript文件中，后者包含了应用程序的所有代码。

❸ 开发者可以使用HTML文件中定义的静态结构，或者Java代码中定义的动态结构，来实现用户界面的布局，这一点我稍后会更加详细地进行解释。对于我们的应用程序而言，该HTML框架为主用户界面页面定义了一个静态结构。最简单的实现方法是使用HTML语言中的表格功能。（正如在Python代码中看到的，我们也可以使用CSS的浮动框来实现，但是，如果想要做动态布局，那么让GWT处理会更好。）因此，我们设置了一个两列的表：一列用于文本输入框，另一列用于“发送”（Send）按钮。

❹ HTML静态结构不仅包括静态结构，还包括静态内容。像往常一样，如果能够将静态内容从程序逻辑中分离出来，我们就应该这样做。因此，我们在这里使用静态帧插入一个标题行，然后使用HTML的表格布局控件，使该标题行跨越表格布局中的两列。

❺ 接下来的内容非常有趣。我们在这里做的是在用户界面里创建一个空区域。*<td>*标签在HTML布局中创建一个区域，但它是空的——标签内没有任何东西。在Java代码中，我们插入内容，然后通过`id=`标签来引用它。我们创建了两个空区域，一个用于文本框，一个用于按钮。

现在，我们分析一些代码。正如在上面的模块声明中看到的，应用程序有一个入口点。完整的入口点方法相当长：既包括用户界面元素的创建、设置事件处理程序，也包括建立客户端/服务器通信的远程过程调用。让我们分部分来看一下，首先从构建主界面部分开始——也就是提醒用户输入他们的名字的主页面：

workspace/HelloChat/src/com/pragprog/aebook/hellochat/client/HelloChat.java

```
❶    public void onModuleLoad() {
❷          final Button sendButton = new Button("Send");
          final TextBox nameField = new TextBox();
```

9

```
        nameField.setText("GWT User");

        // 我们可以为部件添加样式名称
❸       sendButton.addStyleName("sendButton");

        // 给 RootPanel加入nameField与sendButton
        // 使用RootPanel.get() 获取 the entire body element
❹       RootPanel.get("nameFieldContainer").add(nameField);
       RootPanel.get("sendButtonContainer").add(sendButton);

            // 当加载程序时，将光标焦点定位在文本输入框
❺           nameField.setFocus(true);
            nameField.selectAll();
```

❶ 入口点类相当于GWT下的main函数的容器。从概念上讲，它确实像是非图形用户界面工具中的主程序。但在Java中，因为一切都需要封装在类里，所以，必须围绕实际的main函数创建一个框架类。在典型的GWT应用程序中，这是定义入口点类的唯一方法——它只是对一个简单方法进行的过于复杂的包装。真正的main函数是入口点的onModuleLoad方法，顾名思义，这是当GWT模块被客户端加载时，真正要执行的内容。在此方法中，我们创建用户界面部件，对其进行布局，并设置事件处理程序。

❷ 我们在onModuleLoad内做的第一件事情是创建用户界面部件。一般情况下，其构建方式看起来很像在构建一个非浏览器的用户界面。我们创建了一个按钮和一个文本框，用户将在文本框里输入他们的名字。

❸ 这是第一个看起来与传统的非浏览器用户界面不同的地方，该代码行管理部件样式属性。在典型的图形用户界面工具集里，对于不同的样式属性，会有一组相应的调用方法。例如，在Mac OS的Cocoa部件中，我们可以使用像[button setGradientType: NSGradientConcaveWeak]的调用来修改按钮的倾斜度。在GWT中，这些都是用CSS实现的：通过设置CSS属性以创建一个倾斜的图片作为按钮背景，我们将在.gwt-Button CSS样式块中增加一行background: url("images/gradient.png")。在这里，管理样式的唯一的调用就是用来设置与CSS样式的连接。样式名称被GWT转换为一个CSS的class=属性。刚开始它看起来可能有点奇怪，但是用起来确实不错，它有助于保持关注点的分离——开发者确实不应该将代码与视觉样式的内容混杂在一起，而应该将所有样式放在同一个地方。GWT使用CSS的方式为开发者提供了一种真正便捷的方式实现了代码与样式的分离。

❹ 现在，我们到了布局部分。GWT提供了一个图形用户界面的上下文，基本上就是浏览器页面的内容，称为RootPanel。要直接访问根面板，调用RootPanel.get()即可。我们也可以使用HTML实现一部分布局，如果应用程序的主HTML页面包含用id=属性命名的元素，我们就可以使用get(name)访问那些元素。在这个例子中，应用程序的根页面确实提供了应用程序各部分的元素，这是非常典型的GWT风格：我们要在静态布局（通过在HTML中完成）和动态布局（通过用Java编写布局代码）之间做一个选择。通常，布局几乎是固定的（像在这个例子里），写一个HTML表格并用Java来填写表格的内容很容

易。而要创建动态布局，像我们几分钟后会看到的对话框，就需要使用GWT的布局管理器。在静态布局中，我们可以调用get在页面上放置一个布局区域，然后使用add(widget)在其中插入图形用户界面中的部件。

❺ 最后，当加载用户界面时，我们希望它可以正常工作，也就是说，如果用户开始输入，会在文本框中显示相应的内容。我们通过设置*焦点*（focus）实现该功能，焦点是指在屏幕上接收用户界面事件（如按键）的部件。用户可以通过在部件内部点击鼠标来设置焦点，但是如果焦点只在一个地方，会是一件很麻烦的事。因此，我们将焦点设置为文本输入框。我们还让其自动选择放到该区域的占位符文本，这样用户输入的文本会替换占位符。

这就是图形用户界面的基本结构，我们留了另外两个重要的部分。应用程序将从用户那里得到用户名，将其发送给服务器。服务器将此名称放入一个欢迎消息内，将其发送回客户端，然后显示在弹出的对话框里。我们还需要做的是把客户端/服务器的通信和对话框组合在一起。在下一节，我们将了解客户端/服务器通信。首先，对话框是由GWT的用户界面实现的，但是，对话框并不使用HTML文件的静态布局，而是完全动态的：

workspace/HelloChat/src/com/pragprog/aebook/hellochat/client/HelloChat.java

```
    // 创建弹出式对话框
❶   final DialogBox dialogBox = new DialogBox();
    dialogBox.setText("Remote Procedure Call");
    dialogBox.setAnimationEnabled(true);
❷   final Button closeButton = new Button("Close");
    // 通过访问部件的元素来设定部件ID
    closeButton.getElement().setId("closeButton");
    final Label textToServerLabel = new Label();
    final HTML serverResponseLabel = new HTML();
❸   VerticalPanel dialogVPanel = new VerticalPanel();
    dialogVPanel.addStyleName("dialogVPanel");
    dialogVPanel.add(new HTML("<b>Sending name to the server:</b>"));
    dialogVPanel.add(textToServerLabel);
    dialogVPanel.add(new HTML("<br><b>Server replies:</b>"));
    dialogVPanel.add(serverResponseLabel);
    dialogVPanel.setHorizontalAlignment(VerticalPanel.ALIGN_RIGHT);
    dialogVPanel.add(closeButton);
❹   dialogBox.setWidget(dialogVPanel);

❺   closeButton.addClickHandler(new ClickHandler() {
            public void onClick(ClickEvent event) {
                    dialogBox.hide();
                    sendButton.setEnabled(true);
                    sendButton.setFocus(true);
            }
    });
```

9

❶ 首先，我们需要创建对话框。这是一个弹出窗口，因此它不包含在浏览器框架里。这意味着不能仅仅抓住RootPanel，而需要创建一个独立的部件。GWT为此提供了一个便捷的部件：DiaglogBox（对话框）是一个独立的窗口框架，可以嵌入任何GWT部件——我们只需创建它的内容，并将内容插入到对话框中。因为对话框是一个窗口，所以它有一

个标题栏，我们可以使用`setText`方法设置标题栏内容。

❷ 我们希望用户能够在他们需要的时候关闭对话框，所以，创建了一个“关闭”（Close）按钮，我们稍后将该按钮添加到对话框框架中。像往常一样，可以使用CSS设置部件的属性。在这个例子中，我们直接修改HTML代码。对于给定的任何部件，可以通过调用`getElement()`获取部件相对应的XML元素。然后，设置它的ID，让CSS样式可以使用XML元素的`setId()`方法引用该部件。

“关闭”（Close）按钮后，创建了另一组部件。有一个`Label`（标签），它是嵌入在部件中的不可编辑文本。然后，还有一个有趣的部件——`HTML`部件，它是大块HTML文本的封装。无论`HTML`部件的内部是什么，都会直接显示在用户界面的HTML页面中。这对于嵌入样式的文本非常有用，只使用HTML标记文本的一部分往往比产生相同效果的编程操作容易些。

❸ 现在，我们将放置一系列元素。由于没有静态的HTML框架，所以需要由GWT指定如何放置它们。这里的布局只是很多元素的垂直堆叠。GWT有一个部件做这件事，就是`VerticalPanel`。我们只需要按照顺序向面板中添加用户界面的部件即可。在这里，要注意HTML标记，因为有一些文本我们想要用粗体显示。我们并没有创建一个`Label`部件，然后设置其样式属性使其显示成粗体，而只是将该文本封装到<*b*>标签中。

❹ 我们已经将用户界面元素放置在`VerticalPanel`中。剩下的事情就是告诉对话框，面板应该显示为什么样，我们通过设置该对话框的部件实现该功能。现在，对话框的视觉部分全部完成。开始的时候，该对话框不可见。这样的独立部件，只有等到我们显式地告诉它们可见，它们才会真正出现在用户的屏幕上。稍后将看到，我们可以告诉对话框它应该出现在什么地方，从而使其可见。大多数时候，对话框在浏览器窗口的中央，所以，通过调用`center()`方法就可以使对话框可见。

❺ 设置好基本的用户界面，我们终于可以看看如何在GWT中处理事件了！这几乎和Java的Swing库相同。创建一个处理程序对象，使用`addXXXHandler`方法将其附加到相应的部件上。在这个例子中，我们要附加一个处理程序，当用户点击“关闭”（Close）按钮时，关闭该对话框，因此，我们附加了`ClickHandler`对象。在它的`onClick`方法里使对话框不可见，并启用主页的输入区域。

9.3 GWT 中的远程过程调用

现在我们进入较复杂的部分。

前面提到，AJAX代码在GWT中不是显式编写的。我们编写了称为*远程过程调用*（RPC，Remote Procedure Call）的代码。RPC看起来几乎与正常的方法调用一样，但在后台，它由系统转换成从客户端向服务器发送的请求。RPC的返回值是服务器向客户端发送的响应。

就像任何其他RPC系统一样，GWT有一个客户端和一个服务器端。我们可以分别查看它们的代码，这有赖于GWT的RPC系统将它们串起来构成一体。

如果开发者做过分布式编程，可能并不习惯于使用Google式的RPC。传统上，RPC试图尽可

能像通常的函数调用一样出现。换句话说，如果我们想要提供一个阶乘函数的RPC，该功能的实现看起来像一个传统的函数声明，而且调用看起来也像一个传统的调用。例如，Java有一个本地的RPC层，在RPC层我们通过接口定义一个远程对象，然后就可以调用接口类型对象的方法了。

我们可以定义一个阶乘服务作为Java接口：

```
public interface Fact extends Remote {
    int fact(int n);
}
```

然后，在使用该服务的代码中，我们获得远程接口的句柄，像下面这样直接调用它：

```
int j = f.fact(n);
```

通过这样的封装，在服务器中做好了准备工作，使得该对象可用，这样客户端就可以得到其句柄；在客户端也是同样的，我们将编写持有远程对象的句柄的代码。但是，调用本身看起来就像一个普通的本地调用。

这种方法有一个问题：通信慢。相比较于本地调用，远程过程调用可以很容易地花费两个数量级以上的时间。在传统的调用中，代码将停止运行并处于等待状态，直到得到从服务器返回的响应。这在时间上是一个巨大的浪费，而且会在我们的用户界面中带来令人无法接受的延时。

因此，Google为远程调用提供了异步（asynchronous）或延续传递（continuation passing）的风格。调用本身并不返回任何内容，但是，它需要一个额外的参数，该参数是一个函数，当RPC收到返回值时，会对返回值调用该函数。

例如，假设我们有一个阶乘服务。在传统的风格中，我们要做的是：

```
System.out.println("Foo = " + Math.log(3 * f.fact(n)));
```

而在延续传递风格中，代码如下：

```
f.fact(n, new AsynchCallback<int>() {
        public void onSuccess(int result) {
            System.out.println("Foo = " + Math.log(3 * result));
        }
    });
```

我所做的只是将原来处理RPC返回值的代码封装在一个对象中，然后将该对象作为参数传递给远程调用。代码实现的只是启动这一进程，然后，我的代码就可以去做其他事情了。而在此期间，RPC系统会将该调用转换成消息，发送这一消息，等待服务器发送响应，然后将响应消息转换成结果值。一旦返回结果，回调函数就会被调用，而且也会处理RPC调用的结果。

如果开发者习惯命令式编程，可能会感到有点不习惯。（函数式编程人员一直都是这样做的，即使他们不进行分布式编程。）开发者需要一些时间来习惯这种编程，即使习惯以后，有时看起来也似乎有点别扭。但是，总体而言，RPC不必等待的优势胜过其存在的问题。它使得应用程序可以更好地响应用户，而这才是最重要的。

9.3.1 GWT中的客户端RPC

无论一个工具包多么强大，通信永远是不简单的。我们需要能够处理将参数转换为可传递的

消息的格式，而且需要能够处理由于网络造成的延迟甚至失败。为了处理这些问题，GWT使用了一个非常Google化的短语，即异步RPC。在Google，我们用一些非常独特的、风格化的方式去处理基本问题，比如使用异步响应处理程序进行远程过程调用。这对于没有使用Google风格进行过大量分布式开发的人来说非常陌生。总的来说，GWT使这些基础工作对用户不可见是非常了不起的。然而，在客户端RPC中，GWT不这样做。

GWT最初的开发是为了Google内部使用，为了使这种风格变得自然，Google的工程师花了很多时间。虽然它有一点怪，但着实是解决很多RPC的传统问题的最简单的方法。使用异步RPC，虽然我们不需要在客户端编写多线程代码，但仍然需要编写实时响应更新的代码，即只要更新可用就要对其进行响应，而且在等待响应时不会阻塞。

说这么多背景足够了，是时候开始深入细节了，来看看在GWT中编写RPC都涉及了什么内容。我们的Hello World程序有一个远程调用，它将用户的名字发送给服务器，然后得到一个HTML片段，包含给该用户的问候语。在GWT术语中，那是服务器上的代码所提供的一个服务（service）。在GWT中做的第一件事情是为服务方法编写一个同步接口，在我们的Hello World应用程序中，这就是所谓的问候服务。出人意料的是，问候服务接口是编写在客户端软件包里的。（记住，GWT始终以客户端为中心，客户端是接口的用户，所以它位于客户端软件包。）基本的同步问候服务接口如下所示：

workspace/HelloChat/src/com/pragprog/aebook/hellochat/client/ GreetingService.java

```
package com.pragprog.aebook.hellochat.client;

import com.google.gwt.user.client.rpc.RemoteService;
import com.google.gwt.user.client.rpc.RemoteServiceRelativePath;

❶ @RemoteServiceRelativePath("greet")
❷ public interface GreetingService extends RemoteService {
❸         String greetServer(String name);
  }
```

❶ GWT服务接口可以添加注解，说明它们是如何适应应用程序的URL结构的。应用程序有一个根URL，所有的应用程序地址将以该根URL作为前缀。此注解声明了该服务相对于根URL的路径。因此，如果应用程序是在`http://gwt.appspot.com/foo`，那么这个服务的URL将是`http://gwt.appspot.com/foo/greet`。服务接口应该总是按照此方式声明其相对路径的。

❷ 服务接口声明必须是GWT接口`com.google.gwt.user.client.rpc.RemoteService`的扩展。使用此超级接口，说明我们的接口是为了在GWT应用程序中实现其服务，GWT应将其转化为JavaScript。

❸ 这是同步方法声明。任何一个服务方法的参数必须是`java.lang.Serializable`的扩展或者是针对GWT的变种`com.google.gwt.user.client.rpc.IsSerializable`的扩展。这里要理解的一件非常重要的事情是，GWT使用Java同步接口标记将在RPC中传递的类，并不表示它使用Java序列化。使用`IsSerializable`或`Serializable`仅仅是一个标记，意在告诉GWT它需要生成代码来序列化和反序列化该类型。GWT使用的实际格式离兼容Java的标准序列还差得很远。

方法声明本身完全标准，但它只是个普通的接口方法声明。

除了同步接口，还需要编写异步接口。恕我直言，这里我很迷惑，为什么GWT团队不提供一些自动化的支持。要生成异步接口，只需编写另一个接口，它纯粹是样板下的同步接口。异步接口必须包含与同步接口完全相同名称的方法，每个方法必须返回void类型，每种方法在末尾必须增加一个参数，该参数是AsyncCallback，其类型参数是同步方法的返回类型。例如：

workspace/HelloChat/src/com/pragprog/aebook/hellochat/client/ GreetingServiceAsync.java

```
package com.pragprog.aebook.hellochat.client;

import com.google.gwt.user.client.rpc.AsyncCallback;

/**
 *GreetingService的异步接口
 */
public interface GreetingServiceAsync {
        void greetServer(String input, AsyncCallback<String> callback);
}
```

异步接口与同步接口没有显式联系，也没有使用任何特殊的注释，或从任何特殊类继承。异步接口确实仅供客户端使用——真正地在异步和同步接口之间实现映射的基础工作是由GWT生成的。异步接口的唯一目的是给客户端提供将要使用的调用接口。

9.3.2 GWT中的服务器端RPC

GWT中RPC的服务器端非常简单。我们只需要使用一个RemoteServiceServlet的扩展类，就能实现同步客户端接口。对于我们的问候服务，其实现如下：

workspace/HelloChat/src/com/pragprog/aebook/hellochat/server/GreetingServiceImpl.java

```
package com.pragprog.aebook.hellochat.server;
import com.pragprog.aebook.hellochat.client.GreetingService;
import com.google.gwt.user.server.rpc.RemoteServiceServlet;

/**
 *RPC服务的服务器端实现
 */
@SuppressWarnings("serial")
public class GreetingServiceImpl extends RemoteServiceServlet implements
                GreetingService {

        public String greetServer(String input) {
                String serverInfo = getServletContext().getServerInfo();
                String userAgent = getThreadLocalRequest().getHeader("User-Agent");
                return "Hello, " + input + "!<br><br>I am running " + serverInfo
                                + ".<br><br>It looks like you are using:<br>"
                                + userAgent;
        }
}
```

注解`@SuppressWarning`有点不寻常。这个注解之所以放在这里，是因为在后来的Java虚拟机版本中，序列化会为每个类文件生成一个版本标识字段。如果我们有一个实现`java.lang.Serializable`的类，并且不提供版本标识符域，Java则生成一个警告。因为`RemoteServiceServlet`从标准的servlet类继承，而servlet实现了`Serializable`，所以，服务的实现总要实现`Serializable`。但是，由于GWT不使用版本标识符，在代码中包含版本标识符只会引起无谓的混乱。该注解用来防止编译器生成令人费解的的警告消息。

这就是我们需要在服务器端做的所有工作。当然，想要使用数据仓库实现持久性时，会更加复杂。我们将此内容留到下一章中。

综上所述，现在我们已经为这个简单的应用程序实现了完整的远程过程调用。这比在Python中使用AJAX的方式更简洁：我们有定义好的接口，并且可以只通过方法调用来调用该接口。不需要担心什么创建`XMLHttpRequest`、解析参数、检查状态代码，或者任何我们在Python中做的混乱的、容易出错的事情。

9.4　使用 GWT 进行测试和部署

现在已经有了一个基本的GWT应用程序，我们来运行该程序。就像在Python中一样，有两种在App Engine中运行GWT应用程序的方式：本地模式和部署模式。本地模式中，应用程序在开发者的机器上运行；部署模式中，应用程序在App Engine云中运行。

GWT的本地模式和Python的本地模式非常不同。在Python中，本地模式下的测试很友好，不需要部署，但是，并没有真正添加能够支持调试的方式。但使用GWT，在本地模式下，客户端和服务器都运行在Java中，我们可以使用Eclipse的所有功能调试GWT应用程序。这就形成了巨大的差异：我们拥有了断点、踪迹、单步，以及所有其他Java调试工具的完整支持。

要使用Eclipse在本地模式下运行，到“运行”（Run）菜单下，选择“作为网络应用程序运行”（Run As .../Web Application）。GWT将打开一个本地模拟的浏览器环境，用以显示客户端，并启动本地的Tomcat网络服务器执行服务器端代码。

要将应用程序部署到App Engine中，需到包浏览器视图中，右击该项目。在出现的项目菜单上，有一个Google子菜单。只需选择“部署到App Engine”（Deploy to App Engine），开发者的程序就会被放到App Engine云中。如果有任何该项目所需要的有关信息，它都会在第一次运行部署命令时提示开发者填写。

我们完成了第一个基本的GWT应用程序。虽然还没有实现Python聊天应用程序的全部功能，但它有了一个更好的用户界面和一个更清晰的通信层。在下一章中，我们将使用所学到的GWT知识来构建聊天应用程序的Java版本。在这个过程中，我们将了解Java接口的数据仓库，以及GWT放置在服务器端的Java代码的局限。

第 10 章

管理服务器端数据

10

在前面的章节中，我们研究了App Engine中一个基本的GWT应用程序的组成部分。我们构建了图形用户界面，建立了简单的远程过程调用，并且把所有这些基础工作串了起来，使应用程序能够运转起来。在这一章中，我们将使用GWT来实现聊天程序。我们不会花太多的时间讲述如何构建图形用户界面，有关使用GWT构建图形用户界面有成批的书籍，GWT自带的用户界面类的文档也都讲解得非常出色。我们将把大部分时间花在数据仓库（datastore）上，该机制使我们能够在App Engine中使用持久性数据。就像在Python中一样，我们需要做一些额外的工作，使类具有持久性，并且可以查询。本章我们来了解如何在Java中完成这些工作。

我们还将涉及在App Engine环境下使用Java做服务器端基础工作的一些其他问题。具体地说，App Engine对于可以执行什么样的代码，以及代码在云环境中如何运行施加了一些限制，我们将探讨这些限制是什么，以及这些限制对我们编写App Engine应用程序的服务器端有何影响。

10.1 Java 中的数据持久性

如果读者返回去看一下第4章，就会记得，我们需要在Python代码中做一些额外的工作来存储数据，使程序在云中可以正确运行。在Java中，需要做同样的工作。不幸的是，Java的静态类型使得这件事情变得更为繁琐一点。App Engine中使用的基本的后端数据仓库和在Python中的完全一样，但是，要让它在Java中工作，需要考虑的内容会更多。

当然这并不难，但我们需要在代码中放入更多样板。Eclipse照例可以为我们处理很多事情，但是让我们先在没有Eclipse的帮助下开始。我们将手工编写所有的代码，以便于理解其中所有的细节。

在典型的Google风格中，App Engine为了能在Java中使用数据仓库，其所做的工作是截获标准的Java API，即Java数据对象（JDO，Java Data Objects），并挑选出一部分有用的功能。JDO是一个非常复杂的、臃肿的API（如同其他标准一样）。但是，JDO有一个很好的内核，App Engine使用的就是该内核。

下面，我们将看到JDO持久性工作的原理，即它是如何描述持久性对象，以及存储、查询和取回这些持久性对象的。

存储 Java 类

在Python的数据仓库接口中，我们通过给类对象增加属性创建一个持久类。在Java中，我们要做类似的事情，只是在Java中，附加属性要通过使用类声明中的注解添加。通过例子就可以轻松地说明这一点。下面我们将把聊天消息转化为一个数据对象：

workspace/PersistChat/src/com/pragprog/aebook/ persistchat/ChatMessage.java

```
package com.pragprog.aebook.persistchat;

import java.util.Date;

import javax.jdo.annotations.IdGeneratorStrategy;
import javax.jdo.annotations.IdentityType;
import javax.jdo.annotations.PersistenceCapable;
import javax.jdo.annotations.Persistent;
import javax.jdo.annotations.PrimaryKey;

import com.google.appengine.api.datastore.Key;

❶ @PersistenceCapable(identityType = IdentityType.APPLICATION)
public class ChatMessage {

    public ChatMessage() {
    }

    public ChatMessage(String sender, String msg, String chatname) {
        this.senderName = sender;
        this.message = msg;
        this.chat = chatname;
    }

❷   @PrimaryKey
    @Persistent(valueStrategy = IdGeneratorStrategy.IDENTITY)
        private Key key;

❸   @Persistent
        protected String senderName;

    @Persistent
        protected String message;

    @Persistent
        protected String chat;

    @Persistent
        protected long date;

    public Key getKey() {
        return key;
    }
```

```
        public String getSenderName() {
            return senderName;
        }

        public void setSenderName(String senderName) {
            this.senderName = senderName;
        }

        public String getMessage() {
            return message;
        }

        public void setMessage(String message) {
            this.message = message;
        }

        public String getChat() {
            return chat;
        }

        public void setChat(String chat) {
            this.chat = chat;
        }

        public long getDate() {
            return date;
        }

        public void setDate(long date) {
            this.date = date;
        }
    }
```

让我们来仔细分析一下这个例子。

❶ 对于一个要存储在App Engine数据仓库中的Java对象，它的类必须声明为持久性的。在JDO中，开发者可以通过附加一个`@PersistenceCapable`注解来实现类的持久性声明。在完整的JDO中，我们需要声明一个实体类型。App Engine只支持`IdentityType.APPLICATION`，但是，由于Java的类型系统需要注解，以匹配其声明，所以必须声明该字段。

❷ 为了能够存储、取回或搜索某一特定对象，对象必须有一个唯一的关键字以供识别。在Python中，当定义持久性对象时，框架会自动添加一个不可见的关键字段，让数据仓库使用。在Java中，所有内容都需要静态声明。因此，我们需要手动在源代码中插入关键字声明，并将其注解为关键字。我们使用`@PrimaryKey`注解将其标识为关键字，同时需要说明，比较主键的方式是通过对象实体，使用`@Persistent`注解的`valueStrategy = IdGeneratorStrategy.IDENTITY`属性。

大多数时候，我们会使用像这样的关键字对象，它是自动初始化的，并且通常不会直接使用。我们稍后将看到，可以做一些关键字的定制，但通常没有必要。

❸ 持久性对象中应存储的每个字段都需要用`@Persistent`注解。

在Java中，数据对象有一些限制。数据对象大多要求保持对象指针的可维护性。

- 当一个持久性对象包含另一个持久性对象作为字段时，它拥有该对象，并且不允许其他持久性对象引用它。这意味着，我们有时需要保存对象。
- 当一个持久性对象包含其他持久性对象的集合时，它拥有该集合中的所有对象。
- 我们局限于基本的Java集合类，不能用数组，也不能使用任何扩展的集合类型。我们可以用具体的类，如`ArrayList`、`LinkedList`、`HashSet`、`TreeSet`、`Stack`以及`Vector`，也可以使用更抽象的接口，如`List`，但是在保存或者重新装入一个对象时，我们不能保证恢复的对象的列表类型是相同的（这意味着，我们有可能使用`LinkedList`进行保存，而使用`ArrayList`进行取回）。因为这可能具有相当显著的性能影响，我建议JDO字段都显式使用具体的集合类型。
- 我们可以在数据对象中通过使用`@Persistent(serialized=true)`注解标记该对象，从而使用Java序列化类型。需要注意的是，序列化数据对象的行为与开发者通常期望的Java行为有点不同。例如，假设我们在一个持久性对象的列表字段有一个序列化对象的两个副本。如果我们保存该对象，然后加载它，这两个副本不能保证是全等的（`==`），而且它们是否`equals()`也依赖于我们的类的`equals()`方法是如何实现的。
- `String`类型的字段不允许超过500字节。我们可以使用数据仓库中为该字段提供的`Text`类存储更长的字符串，但无法基于该字段的值进行查询操作。
- 如果不使用Eclipse，则需要添加一个额外的编译步骤，称为代码增强（code augmentation）。在增强过程中，App Engine中的JDO实现会给基于持久性注解的类添加代码，使它们能够被数据仓库存储和查询。我们需要确保每次重新编译Java源代码时，都会执行增强过程。（Eclipse自动将JDO的增强包括在项目构建中，所以它会负责处理好这一切——这是使用Eclipse的又一个原因！）

10.2　在 GWT 中存储持久性对象

用Python在数据仓库中存储东西极其简单。我们创建一个持久性对象，然后调用它的`put`方法就可以。转眼间就完成存储了！用Java则需要做更多的工作，主要是需要些静态的样板。Java给我们提供了很多优点，但它确实需要更多样板才能完成工作。

为了能够存储和取回对象，我们需要一种称为`PersistenceManagerFactory`的工厂。工厂的创建代价非常高，我们不想每处理一个请求时都重新初始化工厂；相反，先做好工厂的设置，在应用程序加载到App Engine云的服务器上的时候创建它。而且，我们希望工厂创造在一个不错的、集中的地方，以确保任何需要`PersistenceManager`的人都知道在哪里可以找到该工厂的实例。有一个自然的解决方案——单件设计模式。我们将创建一个`singleton`类，用它静态创建`PersistenceManager`的单一实例，然后，该实例就可以被任何需要它的人访问了。

workspace/PersistChat/src/com/pragprog/aebook/persistchat/server/Persister.java

```
package com.pragprog.aebook.persistchat.server;

import javax.jdo.JDOHelper;
import javax.jdo.PersistenceManager;
import javax.jdo.PersistenceManagerFactory;

public final class Persister {

  private static final PersistenceManagerFactory pmfInstance =
          JDOHelper.getPersistenceManagerFactory("transactions-optional");

  private Persister() {}

  public static PersistenceManagerFactory get() {
    return pmfInstance;
  }

  public static PersistenceManager getPersistenceManager() {
    return get().getPersistenceManager();
  }
}
```

现在，任何需要使用持久性管理器的客户端代码只需调用`Persister.getPersistenceManager()`即可。使用持久性管理器，我们可以通过调用`PersistenceManager.makePersistent(O)`存储对象`O`，接着可以调用`PersistenceManager.close()`。

事务

与任何人讨论分布式应用程序时，你会不断地听到事务性（transactionality）这个词。其原因如下。

事务性可以防止数据被破坏。如果没有事务性，万一在存储内容的过程中被中断——因为网络故障或某台计算机陷入崩溃——就可能会使数据存储不一致。

例如，假想开发者正在编写一个网上商店。开发者创建了一条订单记录，指示运货部门向客户发货，然后开发者又创建了一条收费记录，告诉银行如何收取该笔货款。如果系统在存储发货记录和存储收费记录的中间崩溃，那么，就可能导致发送了商品，却没有收取货款！

开发者想要这两个步骤是原子性（atomic）的，表示要么两者都成功地储存，要么就都不存储。这种要么全部都存储成功，要么全部都没有存储的原子单元就称为事务（transaction）。

Java 的数据仓库提供了一种将多个存储操作集合为一个单一事务的方法。

这比Python的`o.put()`调用麻烦得多：我们需要设置一个持久性管理器工厂，分配一个

PersistenceFactory，然后完成使用时调用close()。幸好这样做还有很多好处。采用PersistenceFactory的接口提供了对事务的支持。从分配一个持久性管理器起，直到调用其close()方法，我们所做的每件事情都是原子性的——也就是说，要么全部成功，要么全部失败。我们存储的每一个对象，每一次对持久性对象所做的改变，要么全部存储，要么全都不存储。这就是事务的美妙之处，提供了关系数据库的安全性。样例代码看起来似乎很麻烦，但也确实有优势。

我们准备要发布一条新消息了。需要创建一个RPC服务，指示客户端收到新消息要发布时将如何告诉服务器。（同样地，值得指出的是，在GWT中，我们将发布操作实现为一个异步RPC——从程序的操作角度看，这正与它的真实行为相同——而不是在混乱的XMLHttpRequest中进行追查。）我们的RPC服务需要两个方法，一个用于发布新消息，一个用于获取消息。基本接口如下所示：

workspace/PersistChat/src/com/pragprog/aebook/persistchat/client/ChatSubmissionService.java

```
package com.pragprog.aebook.persistchat.client;
import java.util.Date;
import java.util.List;
import com.google.gwt.user.client.rpc.RemoteService;
import com.google.gwt.user.client.rpc.RemoteServiceRelativePath;
import com.pragprog.aebook.persistchat.ChatMessage;

@RemoteServiceRelativePath("chat")
public interface ChatSubmissionService extends RemoteService {
  List<ChatMessage> postMessage(ChatMessage messages);
  List<ChatMessage> getMessages(String room);
  List<ChatMessage> getMessagesSince(String chat, Date timestamp);
}
```

像往常一样，我们将其变成异步接口：

workspace/PersistChat/src/com/pragprog/aebook/persistchat/client/ChatSubmissionServiceAsync.java

```
package com.pragprog.aebook.persistchat.client;

import java.util.Date;
import java.util.List;

import com.pragprog.aebook.persistchat.ChatMessage;
import com.google.gwt.user.client.rpc.AsyncCallback;

public interface ChatSubmissionServiceAsync {
    void postMessage(ChatMessage messages,
                 AsyncCallback<List<ChatMessage>> callback);

    void getMessages(String chatroom,
                 AsyncCallback<List<ChatMessage>> callback);

    void getMessagesSince(String chat, Date timestamp,
                     AsyncCallback<List<ChatMessage>> callback);
}
```

我们使用服务器包中的类实现该功能，这个实现起来非常简单。我们用刚刚实现的Persister得到该操作的PersistenceManager，使消息对象具有持久性，以便将它作为事务的一部分进行保存，然后关闭持久性管理器，便会执行该事务。最后，我们使用另一个服务方法，从服务器直接调用，从而把聊天室的更新消息列表提供给用户。

workspace/PersistChat/src/com/pragprog/aebook/persistchat/server/ChatSubmission-ServiceImpl.java

```
    public List<ChatMessage> postMessage(ChatMessage message) {
        PersistenceManager persister = Persister.getPersistenceManager();
        persister.makePersistent(message);
        persister.close();
        return getMessages(message.getChat());
    }
```

如果我们想将任何其他对象作为与聊天消息相同的事务的一部分进行保存，只需要在调用close()之前添加更多的makePersistent()来存储这些对象即可。

10.3 在 GWT 中取回持久性对象

Python中使用一种类似于SQL的查询语言GQL，从数据仓库检索并取回对象。在Java中，我们要做同样的事情，但是，我们所做的工作是标准的Java持久性框架的一部分，需要使用Java的查询语言，而不是为Python构建的客户端语言。Java数据对象的标准查询语言称为JDOQL。和GQL一样，它看起来很像SQL。

说实话，我们并不真正需要使用JDOQL。如果知道我们要取回的对象的关键字，可以使用PersistenceManager的getObjectById方法来获取该对象。例如，如果x是一个聊天消息对象的ID，我们可以用这种方式取回对象x：

```
PersistenceManager pm = Persister.getPersistenceManager();
try {
    pm.getObjectByID(ChatMessage.class, x);
} finally {
   pm.close();
}
```

显然，上述代码的缺陷在于，我们必须要知道关键字。在有些特定的情况下，要取回一个对象，但是我们既不知道关键字，也不能指出它是什么。（我们将在第13章中学习如何解决这种问题。）这里需要做出权衡：通过关键字获取对象比通过查询语句获取的速度要快得多。但是对于我们的聊天应用程序（以及许多类似的应用程序）而言，执行查询语句的速度与封装请求的网络通信代价相比，实在是太小了，甚至可以被忽略。所以，我们现在在聊天应用程序中想取回对象，关键字检索并不是特别有用。然而，对于那些在单个查询过程中就需要进行很多数据仓库交互的应用程序，权衡的结果会完全不同，通过关键字获取对象将会极大地影响到性能。类似于这样的权衡在云编程中比比皆是，开发者需要从诸多因素中找出对性能至关重要的影响因素。

在上述代码中，有一个重要部分：try...finally。前面提到，持久性对象不是轻量级的。持久性对象有很多关联的资源，它挂起的时间越长，就会积累越多的资源。开发者必须确保关闭持久性对象，以便回收资源。如果没有使用try...finally，那么一旦pm.getObjectById(…)和close()之间的任何代码遇到了错误或抛出异常，close()调用就会被跳过，这样的结果会非常糟糕。这不止是存储的问题，更是取回的问题，因为当开发者开始一个存储事务时，通常知道他所要存储的所有内容，因此几乎不会做出任何可能产生错误的计算。如果开发者进行了计算，就不会希望通过close()提交该事务了！但是在取回时，开发者经常会做重复的事情：取回一个对象，然后获得希望识别的其他对象的信息，再取回其他对象，这就可能导致错误发生。因此，try...finally块的安全性对于取回操作很重要。

大多数时候——特别是对我们的聊天应用程序而言——我们将使用JDOQL查询语句描述想要从数据仓库取回什么内容。要取回一个特定聊天室的所有聊天消息，JDOQL查询语句如下所示：

```
select from ChatMessage
where chat=desiredRoom
parameters String desiredRoom
order by date
```

我们的JDOQL查询语句（事实上，大多数JDOQL查询语句）有以下4个部分。

- select from ChatMessage
  select子句声明查询语句要搜索的对象集合。该子句看起来像SQL查询的select子句，并且完成的任务也基本相同。SQL查询语句选择匹配某个过滤器的表行，而select子句则说明从什么表中选择。JDOQL查询语句选择匹配某些过滤器的一组对象，而select子句则说明从什么类中选择。如果要取回ChatMessage的集合，就从ChatMessage类中选择。
- where chat=desiredRoom
  where子句与SQL中的where很像：它提供了一个谓词（即，对我们要取回的对象为真的表达式）。我们希望从一个特定的聊天室取回消息。要取回消息的聊天室的实际值是名为desiredRoom的参数。
- parameters String desiredRoom
  parameters子句在SQL中没有任何对等子句，但是它应该存在。在大多数SQL库中，必须使用一些非常痛苦的、笨拙的语法来声明参数。在JDOQL中，parameters子句声明了一个类型变量的列表，在查询字符串中使用这些变量的地方都会被查询调用中的参数值所替换。因此我们说desiredRoom是一个String类型的参数。
- order by date
  order by子句同样与SQL中的order by相同：它声明了查询语句所选定的对象应该按什么顺序返回。我们希望按照消息发布的顺序看到聊天消息，所以按日期排序。

JDOQL还有另一种语法：不使用查询字符串，而是通过编程实现该功能。我们可以使用下面的代码。

workspace/PersistChat/src/com/pragprog/aebook/persistchat/server/ChatSubmission-ServiceImpl.java

```
@SuppressWarnings("unchecked")
public List<ChatMessage> getMessages(String chat) {
        PersistenceManager persister = Persister.getPersistenceManager();
        try {
                Query query = persister.newQuery(ChatMessage.class);
                query.setFilter("chat == desiredRoom");
                query.declareParameters("String desiredRoom");
                query.setOrdering("date");
                return (List<ChatMessage>)query.execute(chat);
        } finally {
                persister.close();
        }
}
```

这段代码相当简单。@SuppressWarnings是Java处理类型列表方式的神器，因为Java使用称为类型擦除的技术以简化类型集合的编译，所以，无法验证一个类型列表的转换是否有效。由于编译器不能保证开发者不犯错误，因此，它会生成一个警告，让开发者知道，有可能有一个错误。而SuppressWarnings注解就是告诉编译器："闭嘴，我知道我在做什么！"除了这点很小的变化，这一段代码与我们上面所看到的查询语句实现的功能是相同的，只是被转换成了编程的形式。最好采用这种方式，因为它将查询的不同要素分开，使代码更容易理解。

10.4 将客户端和服务器粘合在一起

现在，我们只需要告诉App Engine如何将客户端和服务器端的代码粘合在一起，使聊天应用程序的客户端可以调用RPC方法，并允许它从数据仓库中存储和取回消息。使用App Engine的web.xml就可以完成这项工作，web.xml位于我们的App Engine项目的war/WEB-INF目录下。web.xml文件声明了我们应用程序的servlet部分，然后告诉App Engine将这些servlet设置在服务器端的位置，并告诉GWT如何找到它们。

workspace/PersistChat/war/WEB-INF/web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>

    <!-- Servlets -->
❶    <servlet>
    <servlet-name>chatServlet</servlet-name>
      <servlet-class>com.pragprog.aebook.persistchat.server.ChatServiceImpl
            </servlet-class>
    </servlet>
```

```
❷     <servlet-mapping>
        <servlet-name>chatServlet</servlet-name>
        <url-pattern>/chat/chat</url-pattern>
      </servlet-mapping>

      <!-- 默认服务页面-->
      <welcome-file-list>
        <welcome-file>Chat.html</welcome-file>
      </welcome-file-list>

    </web-app>
```

❶ 告诉App Engine，为了运行应用程序需要在服务器部署哪些servlet。每个servlet有一个名字，与实现它的类相关联。

❷ 针对`servlet`子句所声明的每个servlet，向App Engine提供相应的URL，该URL就是应用程序运行servlet的位置。

在这一章中，我们构建了聊天应用程序的基础设施：有了客户端接口，可以通过RPC发布新聊天消息和取回信息。我们建立了RPC方法的servlet实现，并使用了App Engine JDO接口来访问数据仓库。通过这些工作，我们学习了如何从数据仓库存储和取回对象的基础知识。

下一章我们将返回到用户界面，看看如何使用刚刚建立的RPC服务为聊天应用构建一个非常漂亮的图形用户界面。在实现过程中会探讨可用的图形用户界面工具。我们将涉及如何在GWT中创建图形用户界面布局，如何在用户界面中不用进行完整页面刷新就可以更新数据显示，以及如何响应用户操作。

最后，我们来了解一些使用数据仓库的更复杂的方式。开发者可以使用数据仓库做很多非常有趣的事情，但是，也受到了一些比较特殊的限制。我们将深入研究数据仓库的功能和局限性。

10.5　参考文献和资源

- Java数据仓库API，http://code.google.com/appengine/docs/java/datastore/。
  App Engine Java数据仓库的官方文件。
- Java数据对象，http://java.sun.com/jdo/。
  JDO标准文档。JDO是Java到App Engine数据仓库的接口所依赖的基本技术。
- “Java 持久性 API”——一个实体持久性的简单编程模型，http://java.sun.com/developer/technicalArticles/J2EE/jpa/。
  概述App Engine中使用Java持久性API的一篇文章。

第 11 章 用Java构建用户界面

现在，大家已经了解了如何在Google App Engine中使用Java工作的一些基本知识。我们学会了如何使用数据仓库进行持久性数据存储的基础知识，研究了如何使用GWT构建一个完整的Java云应用程序的通用结构。在本章中，我们将更详细地了解如何使用GWT工作，构建云应用程序的用户界面。

11.1 为什么使用 GWT

在深入之前，值得一提的是，开发者并不是非得需要使用GWT构建采用Java的App Engine服务。开发者可以使用任何想要使用的Java网络应用程序开发框架——只要遵循App Engine运行时所施加的一些限制即可。（在App Engine上运行的Java程序不能使用线程、锁或可运行对象，这些都是受限的。）

但是开发者可以使用大多数常见的基于Java的网络框架。开发者可以使用标准的servlet环境，或者Struts，甚至Grails。GWT并不是App Engine的不可或缺的部分，那么，为什么我要关注它呢？

一部分原因只是因为我喜欢它。我用过很多不同的工具包构建网络和基于云的应用程序。以我的经验，GWT是最好的。开发中最痛苦的地方，即最容易出问题的地方，都可以由GWT中行之有效的标准代码发生器自动生成。

再就此问题详细说明一下。GWT为开发者做的是，使为云应用程序编写的用户界面和为普通桌面应用程序编写的用户界面一样自然。GWT为开发者提供了用于测试和调试的工具，即使开发者的代码使用了不同的语言，也很容易调试——就好像是开发者的Java服务器和客户端用户界面，实际上是用JavaScript或者HTML 5运行的。

这就是GWT最有价值的功能。我们知道，云编程最大的问题往往是它测试和调试起来很痛苦，而且很难。在云应用程序中，开发者需要处理客户端程序（HTML+JavaScript），加上实现服务器端所使用的编程语言，再加上XML和HTTP，这已经非常复杂，但是还不止于此。开发者不仅需要处理这些语言，而且还需要处理各种语言之间的边界。开发者不仅需要编写客户端上的JavaScript代码，而且所编写的JavaScript代码要实现各种功能，包括如何生成HTTP请求，如何解析XML响应，以及如何生成JavaScript对象。开发者的服务器端不仅需要有执行基本操作的代码，而且需要知道如何解析HTTP请求，以找出客户想要它执行什么操作，同时，它还需要获取这些

操作的结果，并翻译成XML。客户端和服务器端需要就请求、响应和XML编码是什么达成绝对的一致。任何一段代码，尤其是语言之间边界区域的代码，都可以导致微妙且难以发现的问题。我曾把很多时间都浪费在追查由XML格式或HTTP消息头中的细小失误所带来的错误上。

GWT最重要的地方，并不在于它可以让开发者像使用本地用户界面套件似的部件来编写用户界面。当然，那是非常友好，而且很有价值的，但是更重要、更有价值的是，GWT能处理各种不同的语言：开发者用Java编写所有代码，GWT负责将Java转换成XML、HTTP、JSON、JavaScript、HTML和CSS，并且生成跨越语言的代码。在GWT的掩盖下，底层的所有东西仍然保持原状——但是通过GWT，开发者不需要直接处理底层。开发者不会因为各个语言的差异而产生问题，这样所节省的时间和精力是非常惊人的。

11.2　使用部件构建 GWT 用户界面

现在开始使用GWT来构建用户界面。我们将在聊天应用程序上构建一个基本用户界面，它与我们前面用Python所构建的用户界面相同，只是这次采用的是GWT。首先，我们集中研究真正的用户界面方面——除某些必要之处，我们会忽略应用程序的逻辑，而只专注于应用程序的呈现方面。这是开发者构建基于GWT的App Engine服务的典型方式：首先要搞清楚想要服务做什么。正如我们在前面的章节中所做的那样，先找出想要管理的基本数据，以及需要什么类型的RPC调用，然后，再坐下来使用GWT组建用户界面。

和很多用户界面工具套件一样，在GWT中，开发者使用*部件*（widget）进行工作。部件是用户界面的基本元素。GWT提供了开发者惯用的所有通用用户界面元素部件：文本框、按钮、单选按钮、下拉菜单、菜单，等等。也有用于管理布局的*容器部件*（container widget）。如前所述，这种整体结构是我们使用GWT的一个重要原因，因为开发者不必担心如何编写CSS实现其用户界面布局的问题，也不必弄清楚如何设置所有的JavaScript来绘制其用户界面。开发者使用部件构建其用户界面，包括容器部件，然后由GWT来完成剩下的工作。开发者可以专注于“做什么”，而不是“如何做”。

在传统的图形用户界面工具套件中，开发者从窗口开始，然后在窗口中放置部件。使用GWT，开发者将其应用程序放在一个网页中，然后把部件放入其中。所以在GWT中，出发点是一个HTML页面。开发者可以把任何他想要的HTML代码放到这个基本页面中。事实上，只要开发者想，他就可以用HTML实现几乎所有的图形用户界面布局。早在第9章，我们就用HTML完成了基本布局。不过实际上，开发者没有必要这样做，他可以使用一个完全空白的HTML页面，并且只使用GWT就能完成所有布局。这就是我们将在本章做的工作。我们仍然需要一个HTML页面，因为加载云应用程序的唯一方式就是通过网页。因此，我们将使用一个结构最少的页面，如下面代码所示。它只是设置加载GWT用户界面的链接，使用<link>标签加载CSS，使用<script>标签加载用户界面代码。

先从HTML页面开始。为了告诉App Engine哪个页面是应用程序的主框架，我们按照与上一章相同的方式编辑`web.xml`文件。我们喜欢将应用程序的主页面称为`Chat.html`，因此，只需编

辑welcome-file-list条目，如下：

```
<welcome-file-list>
    <welcome-file>Chat.html</welcome-file>
</welcome-file-list>
```



```
<welcome-file-list>
    <welcome-file>Chat.html</welcome-file>
</welcome-file-list>
```


当系统获知了哪个页面提供应用程序框架之后，我们就可以通过创建Chat.html页面设置该框架了：

workspace/PersistChat/war/Chat.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

<html>
    <head>
        <meta http-equiv="content-type" content="text/html; charset=UTF-8">

        <link type="text/css" rel="stylesheet" href="Chat.css">

        <title>AppEngine Chat</title>

        <script type="text/javascript" language="javascript"
                src="persistchat/persistchat.nocache.js">
        </script>
    </head>

    <body>
    </body>
</html>
```

上面的代码没有太多内容，它是一个非常小的文件。主体部分完全是空的。我们将完全使用GWT的动态布局代码来填充该文件，重要的是如下内容。

- 一个指向应用程序的.css页面的链接。开发者通常会给CSS和HTML文件使用相同的名字，因此，这里的CSS文件就是Chat.css。
- 一个脚本标签，用来加载GWT生成的JavaScript代码。

有了该页面作为聊天应用程序的框架，我们就可以真正开始工作了。这个聊天用户界面里该有什么内容呢？让我们来看看此之前的图7-2中组装的样子。顶部有一个标题栏；在标题栏下，有一个包含当前时间和日期的子标题；然后，有一个包含聊天室列表的区域，以及一个与之并排的含有聊天消息的区域。这些区域下面，是一个我们可以输入新消息的输入框，以及发送消息的按钮。

在GWT中，我们可以按照刚才描述的内容来设置用户界面。用户界面的基本结构是垂直布局：顶部区域有标题和子标题，中间区域有聊天室列表和聊天消息，底部区域包含文本输入。我们希望通过编写GWT代码设置这些部件，并将其安排在用户界面中。

在GWT中，客户端应用程序总是从调用应用程序主类的onModuleLoad方法开始的。onModuleLoad基本上就是一个GWT应用程序的主函数，而且，因为GWT应用程序需要做的第一件事情就是设置其用户界面，所以应将用户界面的构造代码放在这里。在本节的其余部分，我们再来实现onModuleLoad。

先从创建用户界面的基本框架开始：

workspace/PersistChat/src/com/pragprog/aebook/persistchat/ client/Chat.java

```
❶ final VerticalPanel mainVert = new VerticalPanel();

❷ final VerticalPanel topPanel = new VerticalPanel();
  final HorizontalPanel midPanel = new HorizontalPanel();
  final HorizontalPanel bottomPanel = new HorizontalPanel();
❸ mainVert.add(topPanel);
  mainVert.add(midPanel);
  mainVert.add(bottomPanel);
```

❶ 为了使用一系列层叠的用户界面元素来设置用户界面，我们必须创建一个Panel（面板）部件，然后将其他元素插入该部件中。首先创建一个VerticalPanel，这是用户界面的主要元素。

❷ 现在，要创建用户界面的垂直元素。我们需要三个部件。在顶部，需要垂直层叠一个标题栏和一个副标题，所以，这是另一个VerticalPanel；在中间，我们要有两个并排的部件（聊天室列表和聊天记录区），所以，这是一个HorizontalPanel；最后，底部还要加上输入框和发送按钮，它们同样也是垂直层叠的，因此，又有一个VerticalPanel。

现在，我们要把标题和副标题放到顶部的面板中：

workspace/PersistChat/src/com/pragprog/aebook/persistchat/client/Chat.java

```
❶ final Label title = new Label("AppEngine Chat");
  final Label subtitle = new Label(new Date().toString());
❷ title.addStyleName("title");
❸ topPanel.add(title);
❹ topPanel.add(subtitle);
```

❶ 在顶部面板中，我们想要放置两个部件，每个都只显示一些文本。在GWT的用户界面中加入文本的最简单的方式是使用Label（标签）部件。我们为标题和子标题都创建了标签部件。因为标题是被动且不会改变的，所以，我们可以在创建它们时就将文本放到其中。

❷ subtitle部件的内容只是纯文本。但对于标题而言，我们要使它看起来有些不同，标题中的文本应该比其下方文本更大、更粗。

大多数用户界面套件提供一些管理样式的方法，以设置文字大小、字体、颜色等。事实上，我们已经知道了在网络环境中的设置方式，CSS正是一个非常友好、灵活的管理样式属性的方式。为什么要推倒重来呢？GWT并不定义新的API来实现样式，而是使用CSS。开发者不需要编写太多代码，GWT有默认样式，可以使部件看起来更美观。但当开发者要自定义某些内容时，仍然可以使用CSS声明属性。在应用程序的CSS文件中编写开发者想要的样式的CSS，然后通过调用addStyleName("style")告诉GWT对某个部件使用某

种特定的样式。我们将为标题标签设置apptitle样式。（还有一个setStyle名称的函数，但是，它会清除与该部件相关的所有其他样式属性。GWT能自动设置很多属性，使部件看起来还不错，因此，开发者并不希望清除这些属性。addStyleName只是将开发者的样式增加到CSS级联中，因此它依然继承了所有样式属性，而不是开发者特别设置的样式。）我们稍后会学习在哪里放置CSS的内容。

❸ 创建好各部件后，我们需要将其添加到面板中。

上面的代码调用了addStyle改变包含应用程序标题的标签部件的外观。我们所做的就是插入一个对CSS样式的引用，这就是使用GWT改变任何有关我们的应用程序部件的风格要做的工作。GWT在设置默认值方面确实做得不错，但是在实际的应用程序中，开发者都会有想要定制的地方。CSS就是进行定制的极好方式，它以标准的方式提供了一套完整的样式属性，通过CSS的级联结构，可以很容易地为开发者的应用程序设置通用风格，并在必要的地方进行定制。

接下来，我们改变一些样式，其基本的模式始终是相同的。在Java代码中，我们给想要修改样式的元素增加一个CSS类属性，然后在CSS文件中，为其编写一个类条目。以下是我们向默认的CSS添加的内容：

workspace/PersistChat/war/Chat.css

```
.title {
    font-size: 4em;
    font-weight: bold;
    color: #4444FF;
}

.messages {
    background: #AAAAFF;
}

.emphasized {
    font-weight: bold;
    background: #FFFF88;
}
```

我们实现了3个样式类。我将详细描述第一个，举一反三，其他两个应该就可以理解了。第一个CSS类是应用于刚刚创建的标题标签的。我们想使标题文本变大而且加粗，并且使用彩色背景。因此，在CSS类中，我们修改其font-size属性使之变大，修改其font-weight属性使其加粗，修改其background元素改变其背景颜色。

就这样，我们完成了样式的改变。

我们还需要设置其他两个子面板的内容。基本机制与刚刚看到的相似，但我们使用了一些更有趣的部件。聊天列表面板特别值得关注，因为它的内容是一组基于可用的聊天室动态生成的链接。这有点复杂，所以我们会稍后分析此内容。首先来看看其他子面板的基本布局。下面是中间面板的基本布局代码：

workspace/PersistChat/src/com/pragprog/aebook/persistchat/ client/Chat.java

```
❶ final VerticalPanel chatList = new VerticalPanel();
  chatList.setBorderWidth(2);
  final Label chatLabel = new Label("Chats");
❷ chatLabel.addStyleName("emphasized");
  chatList.add(chatLabel);
❸ chatList.setWidth("10em");
❹ populateChats(chatList);
  //"TextArea text"被定义为类字段，从而文本区域可以被处理方法所引用
❺ text = new TextArea();
  text.addStyleName("messages");
  text.setWidth("60em");
  text.setHeight("20em");
  midPanel.add(chatList);
  midPanel.add(text);
```

❶ 我们要将聊天列表放在左边的列里，因此，创建了一个垂直面板。然后通过将其边框设为2，设置一个可见的边框，并在面板上面放置了一个标签。

❷ 正如以前所做的一样，我们要改变聊天列表上的标签的样式。这一次，将使用`emphasized`样式，其定义可以在上面的CSS中看到。

❸ 通常情况下，GWT根据面板中的最大部件的大小选择默认的面板大小。我们想要多些控制，因此显式地将其宽度设为10 em。10 em看起来似乎是随机的，其实不是。显示聊天记录的文本区域将是60 em，因为这个宽度明显能够适合多数消息。在试验了不同宽度后发现，聊天室列表的宽度是聊天记录的六分之一时能得到最漂亮的外观。如果聊天室列表更窄，列表外观会显得难看、瘦弱；如果聊天室列表更宽，则会在窗口的左侧留下太多空白。

❹ 为了填充聊天室列表中的内容，需要设置一些RPC和回调函数。把内容嵌入在构建基本用户界面布局的地方，多少显得有些笨拙，而且违背了我们关注点分离的原则。用户界面的布局是一个关注点，RPC和事件处理是另一个。因此，我们只调用完成此工作的代码。稍后再来处理这部分实现。

❺ 最后，我们来创建文本区域。设置其宽度为60 em，高度为20 em，然后，添加完所有刚刚创建的面板部件，这样，我们中间部分的布局就全部完成了！

最后，再来放置底部的各部件：

workspace/PersistChat/src/com/pragprog/aebook/persistchat/ client/Chat.java

```
final Label label = new Label("Enter Message:");
label.addStyleName("bold");
final TextBox messageBox = new TextBox();
messageBox.setWidth("60em");
final Button sendButton = new Button("send");
bottomPanel.add(label);
bottomPanel.add(messageBox);
bottomPanel.add(sendButton);
setupSendMessageHandlers(sendButton, messageBox);
```

几乎是相同的工作。先创建部件，再对它们进行布局。同样，我们需要设置一些事件处理程序和回调函数，而且仍要保持关注点分离原则，并且只调用一个设置函数。

现在，我们已经完成了用户界面的布局，布局中唯一剩下的事情是焦点管理。用户界面的焦点就是选择作为任何动作的默认目标的部件。如果用户开始打字，焦点就是接收用户输入的字符的部件。从某些方面看，设置焦点是个小细节，但是，从开发者的应用程序是否流畅运行这个角度来说，确保用户看到的焦点在正确的地方将会产生很好的效果。

下载 workspace/PersistChat/src/com/pragprog/aebook/persistchat/ client/Chat.java

```
RootPanel.get().add(mainVert);
// 设定指针焦点在消息框中
messageBox.setFocus(true);
messageBox.selectAll();
setupTimedUpdate();
}
```

用户运行一个聊天应用程序时，会希望能够开始输入聊天消息，因此，我们要确保让他们输入消息的部件是活跃的。

11.3　激活用户界面：处理事件

现在已经有了一个用户界面，如果我们试着显示一下，它看起来确实不错。但是，该界面没有任何活动性内容。它不知道如何响应用户发生的任何动作，也不知道如何从服务器获取数据。在GWT中激活用户界面的关键是回调函数。

到目前为止，生成的这个用户界面看起来和我们所希望的一样，但它完全是被动的，不做任何事情。为了使用户界面能够实现预期功能，我们需要设置事件处理程序。事件处理程序代码本质上是异步的，完全由回调函数构成。GWT的事件处理程序代码创建的对象，包含有处理特定事件的代码，而不是像Windows中的事件循环那样，在事件循环中，开发者要编写一个循环监测用户接口动作，然后进行决策。

在考虑处理用户动作之前，还需要处理一些其他活动。还有一部分用户界面布局工作没做——那就是活跃聊天室列表。问题是，我们不知道服务器上存在哪些聊天室，所以在填充用户界面时，无法直接生成该列表。我们需要检索到这个列表，并且要在不干扰应用程序显示的条件下完成这项工作。

在App Engine中使用GWT，这个过程很有代表性。我们需要发送一个请求到服务器，获取聊天室列表，但不希望在等待响应时，程序就这样暂停，并且在用户的窗口中不显示任何内容。在一个典型的Java程序中，我们可能会用线程处理该问题——可以创建一个`Runnable`对象，并运行该对象，完成取回和填充聊天室列表的功能。但是，App Engine所提供的是一个功能有限的受控环境，它不允许我们创建线程。

继续传递编程方式

这种基于回调函数的编程方式，有时也称为继续传递风格（CPS，continuation passing style）代码，这个说法来源于函数式编程社区。其基本思想是：任何程序都可以被写成完全异步的风格。当代码调用函数 f 并使用 f 的结果时，开发者都可以换用另一种方法来代替：给 f 再添加一个新参数，这个参数是一个以 f 结果为参数的函数，从而使得当 f 产生结果时，会以 f 的结果再调用该函数。

例如两个数相乘的函数：

```
def mult(m,n):
  return m*n
```

在继续传递风格中，则可以这样写成如下代码：

```
def cpsmult(m, n, done):
    done(m * n)
```

如果开发者想要用 GWT 进行开发，那么，继续传递风格 CPS 将无处不在。用户界面事件处理代码基本上都是 CPS 代码，而且 GWT 中的 RPC 的异步形式也恰恰是过程调用的 CPS 形式。

我们可以使用GWT的异步调用，涉及好几个步骤。需要给包含RPC调用的服务添加一个方法，用来获得聊天室列表。我们稍后会详细地分析如何添加服务获取聊天室列表。然后在onModuleLoad方法中创建一个垂直面板。最后，调用populateChats方法，使用GWT的异步RPC取回聊天室列表，并填充该列表。具体代码如下：

workspace/PersistChat/src/com/pragprog/aebook/persistchat/ client/Chat.java

```
    /**
     *建立一个调用，通过回调来获取可用聊天室列表，
     *当服务器响应时，会创建联接部件并把它们添加到chatListPanel中。
     */
    public void populateChats(final VerticalPanel chatListPanel) {
❶     chatService.getChats(new AsyncCallback<List<String>>() {
❷          public void onFailure(Throwable caught) {
                chatListPanel.add(new Label("Couldn't retrieve chats: "caught));
            }

❸           public void onSuccess(List<String> chats) {
                for (String chat : chats) {
                    Button chatButton = new Button(chat);
                    chatListPanel.add(chatButton);
❹                   Chat.this.setupChatClickHandler(chatButton, chat);
                }
               setCurrentChat(chats.get(0));

            }
        });
    }
```

这是一个RPC和异步操作的相对简单的用法，它是开发者在App Engine/GWT中会反复使用的基本模式。

❶ 生成一个RPC调用。我们并不是一直在活跃地等待响应，等到后就返回，而是设置一个回调对象，它会在RPC的结果可用时调用。这种模式经常被使用，不仅仅是在RPC中。由于缺乏线程支持，几乎在我们会使用典型Java程序的线程的任何地方，在App Engine中都会设置某种回调函数。换句话说，我们这个应用程序几乎所有真正的代码都在回调函数中运行。

❷ 在大多数时候，RPC应该成功。正如读者看到的，我们在编写服务器端代码时，服务器端并不会返回一个错误。但是在云程序中，客户端和服务器之间总是有另外一层——那就是网络。网络层是一个潜在的出错来源，超出了个人控制，所以编程要仔细，一定要考虑周全，提前做好各种准备。在这个例子中，我们将以一种非常简单的方式处理RPC：如果`getChats`调用失败，就会在聊天室列表部件中放入一条错误消息。

❸ 如果RPC按照预期希望执行成功，那么就填充聊天室列表面板。针对每个聊天室，我们创建了一个包含聊天室名称的`Button`构件，并把它添加到面板中。

❹ 然后，我们需要设置事件处理程序。点击某个聊天室按钮时，我们想让用户界面做有所响应，为此，需要设置处理程序。我们将在下一节来看一下如何实现该功能。

为了使应用程序能够响应事件并执行操作，需要设置*事件处理程序*（event handler）。事件处理程序是在用户执行应用程序希望关注的操作时被调用的回调函数。例如在上面的代码中，我们只是创建了一堆用于选择聊天室的按钮，但是这些按钮还没有任何功能。因此，尝试下面的代码：

workspace/PersistChat/src/com/pragprog/aebook/persistchat/client/Chat.java

```
protected void setupChatClickHandler(final Button chatButton, final String
chat) {
    chatButton.addClickHandler(new ClickHandler() {
        public void onClick(ClickEvent event) {
            setCurrentChat(chat);
            text.setText("Current chat: " + chat + "\n");
            currentChat = chat;
            chatService.getMessages(currentChat, new MessageListCallback());
        }
    });
}
```

以上代码给一个按钮附加了一个处理程序，后者是`ClickHandler`的一个实例。点击按钮时，会调用该处理程序的`onClick`方法。也就是说，当用户点击一个特定聊天室的按钮时，该处理程序将把选定的聊天室设置为当前的聊天室，并且它还将调用RPC从选定的聊天室中取回消息。

为使聊天应用正常工作，我们还需要编写更多事件处理程序。当用户试图点击发送按钮发送聊天信息时，应该发生如下事情：系统获取文本输入框中的内容，将其作为一条新的聊天消息发送到服务器，更新聊天记录，并且清除输入区域，为下一条消息做准备。

我们设置一个回调函数，当用户点击“发送”按钮或者在消息输入区的文本上输入回车键时调用它：

workspace/PersistChat/src/com/pragprog/aebook/persistchat/client/Chat.java

```
    private void setupSendMessageHandlers(final Button sendButton,
            final TextBox messageBox) {
        // 为sendButton与nameField创建事件处理程序
❶       class SendMessageHandler implements ClickHandler,
                KeyUpHandler {
            /** 当用户点击sendButton按钮时调用*/
❷           public void onClick(ClickEvent event) {
                sendMessageToServer();
            }

            /**当用户在nameField按下回车键时调用*/
❸           public void onKeyUp(KeyUpEvent event) {
                if (event.getNativeKeyCode() == KeyCodes.KEY_ENTER) {
                    sendMessageToServer();
                }
            }

            /**发送聊天消息给服务器*/
❹           private void sendMessageToServer() {
                ChatMessage chatmsg = new ChatMessage(user,
                        messageBox.getText(), getCurrentChat());
                messageBox.setText("");
                chatService.postMessage(chatmsg,
                        new AsyncCallback<Void>() {
                    public void onFailure(Throwable caught) {
                        Chat.this.addNewMessage(new ChatMessage(
                                "System", "Error sending message: " +
                                caught.getMessage(),
                                getCurrentChat()));
                    }

                    public void onSuccess(Void v) {
                        chatService.getMessagesSince(getCurrentChat(),
                                lastMessageTime,
                                new MessageListCallback());
                    }
                });
            }
        }
❺       SendMessageHandler handler = new SendMessageHandler();
        sendButton.addClickHandler(handler);
        messageBox.addKeyUpHandler(handler);
    }
```

❶ 我们正在编写的是一个将被两种事件调用的处理程序。当用户点击“发送”(Send)按钮时，以及当用户在消息框中按下回车键时，该处理程序都会被调用。我们需要实现处理每个事件的接口：对于按钮点击，有`ClickHandler`接口；对于回车键，有`KeyUpHandler`接口。

❷ `ClickHandler`接口有一个`onClick`方法，任何时候点击按钮，都会调用该处理程序的`onClick`方法。按钮点击处理程序和回车键处理程序做的是同样的事情，所以我们将其抽象为一个函数。

❸ `KeyUpHandler`接口有一个`onKeyUp`方法，该方法在按下键时被调用。主要的区别是`onKeyUp`在按下任何键时都会被调用，但我们只想在按下回车键时发送消息。因此，我们需要做一个测试，以检查什么键被按下，只有在按下回车键时才进行消息发送。

❹ 这里是实际进行处理的地方。首先，我们创建聊天消息，然后，使用惯用的GWT的异步方式，调用GWT的RPC消息。

❺ 最后，我们创建一个回调对象的实例，将其注册为点击“发送”按钮或者在文本框中按下回车键时的事件处理程序。

11.4 激活用户界面：更新显示

至此已经布局好了用户界面，并且构建了回调函数和事件处理程序，使应用程序能够实际做一些事情以响应用户操作。但我们仍然缺少了一个关键部分——更新显示。当用户选择一个聊天室时，用户界面应该被更新，以显示选定聊天室的消息，然后，每当新消息发布时，用户界面应该保持更新。

在上节的事件处理代码中的选择聊天室处理程序中（以及其他一两个地方），我们创建了`MessageListCallback`。这段代码实际上就是用新消息的集合来更新显示。`MessageListCallback`的使用方式有两种。

(1) 当用户选择一个新的聊天室时，调用它来显示新的消息。在这种情况下，它取回该聊天室中到现在为止的所有消息的完整列表。

(2) 为了保持用户界面的更新，我们有一个被定期调用的回调函数，不断地取回新消息：该回调函数不获取聊天室中的所有消息的完整列表，只是获取还没有被该客户端看到的消息列表。（我们将在下一章学习它怎样判断哪些消息已经或还没有被客户端看到。）

更新用户界面确实很容易。基本上，所有用户界面部件都有方法来改变其内容，开发者只需将其转换为字符串，然后将它们添加到部件中。

因此，让我们看看`MessageListCallback`：

workspace/PersistChat/src/com/pragprog/aebook/persistchat/ client/Chat.java

```
public class MessageListCallback implements AsyncCallback<List<ChatMessage>> {

    public void onFailure(Throwable caught) {
    }

    public void onSuccess(List<ChatMessage> result) {
        addNewMessages(result);
    }
}
```

```
protected void addNewMessages(List<ChatMessage> newMessages) {
    StringBuilder content = new StringBuilder();
    content.append(text.getText());
    for (ChatMessage cm : newMessages) {
        content.append(renderChatMessage(cm));
    }
    text.setText(content.toString());
}

protected String renderChatMessage(ChatMessage msg) {
    Date d = new Date(msg.getDate());
    String dateStr = d.getMonth() + "/" + d.getDate() + " " +
        d.getHours() + ":" + d.getMinutes() + "." + d.getSeconds();
    return "[From: " + msg.getSenderName() + " at " +
        dateStr + "]: " + msg.getMessage() + "\n";
}

protected void addNewMessage(ChatMessage newMessage) {
    text.setText(text.getText() + renderChatMessage(newMessage));
}
```

❶ 回调函数本身其实很简单：接收消息列表，将其添加到显示，并且只调用addNewMessages。

❷ addNewMessages的实现几乎一样简单。它将聊天消息作为字符串显示，将包含新消息的字符串加到聊天窗口中已经存在的消息字符串之后，然后设置窗口内容。

还有一部分代码，用来真正激活用户界面，使其按照我们希望的方式工作。我们希望当其他用户发布新消息时，用户界面自动更新。在云环境中说"一旦别人发布消息时就更新"，我们其实并不知道应该如何让其真正地工作，因为客户端不知道别人在何时发布消息。客户端没有办法知道，只有服务器知道，而服务器只能响应客户端的请求。因此，我们创建一个周期性的更新：在定期调度中设置一个会自动发送到服务器端的请求，用来查询是否有任何更新。这是一种在GWT用户界面中非常通用的模式，因此，在GWT中很容易做到这一点。此外，当然，这基本上是另一个回调函数，称为定时器（Timer）。Timer是一个可运行的对象，GWT会在调度时调用。我们创建并设置定时器作为我们的onModuleLoad方法的最后一部分：

workspace/PersistChat/src/com/pragprog/aebook/persistchat/ client/Chat.java

```
private void setupTimedUpdate() {
// 创建一个新的定时器
Timer elapsedTimer = new Timer() {
    public void run() {
        chatService.getMessagesSince(getCurrentChat(), lastMessageTime,
                new MessageListCallback());
    }
};
//每500毫秒调用一次定时器
elapsedTimer.scheduleRepeating(500);
}
```

定时器对象的创建绝对是超级标准的GWT风格的回调函数代码。我们创建一个Timer对象，它被调用时，定时器向服务器发送请求，请求更新。在更典型的GWT代码中，提供了又一个回调函

数——这次，是一个在更新请求返回结果时才会调用的回调函数。当`getMessagesSince`调用返回时，新消息将会通过调用`addNewMessages`显示。

定时器对象被创建之后，我们只需要告诉它应该多久被调用一次。对于一个响应式的用户界面，二分之一秒是一个不错的时间间隔，因此，我们令定时器每500毫秒被调用一次。

11.5　GWT 结束语

在本章，我们组装了聊天应用程序的基本图形用户界面，研究了构建GWT应用程序的基本机制。我们看到了如何创建部件并在用户界面中放置部件，还了解了如何使用CSS定制用户界面的元素的样式。我们研究了如何创建事件处理程序，并将其附加到部件，使用户界面变得活跃起来，能够响应用户操作。事实上，我们已经看到了使用Java构建一个完整的App Engine程序所需要做的一切工作。

下一章将把所有的内容整合到一起。我们将完成服务器端的代码，以及与服务器交互的客户端逻辑的剩余代码。在此过程中，我们能了解更多GWT提供的有趣服务。这些内容应该看起来都很熟悉：GWT使用继续传递回调风格的相当普遍。大多数服务由钩子函数提供，我们可以把自己的回调函数附加到钩子函数上。到下一章结束时，聊天应用程序将全部构建完成，并部署到App Engine中。

11.6　参考文献和资源

- GWT的部件库，http://code.google.com/webtoolkit/doc/1.6/RefWidgetGallery.html。
  对于GWT用户界面构建者最有用的资源。本网站列出了最新的GWT部件，每个部件都有可视化示例，还有关于定制部件所使用的CSS属性的完整说明，并列出编程所使用的事件处理程序。
- GWT 2.0开发者指南，http://code.google.com/webtoolkit/doc/latest/DevGuide.html。
  官方GWT文档，描述开发者可能想知道的关于GWT的一切内容。

第 12 章 构建Java应用程序的服务器端

我们已经构建了聊天应用程序的大部分代码，剩下的就是客户端和服务器端之间的连接。早在第10章，我们就组建了一个基本的RPC接口，用于连接客户端和服务器端。不幸的是，自从建立了客户端以后，整个系统就变得不太正确了。在这一章中，我们将看看在定义该接口时做错了什么，以及如何解决这一问题。我们将更加深入地了解在服务器上到底需要哪类对象和方法，然后会实现缺失的部分。我们会研究除了客户端请求的处理程序之外，在服务器端还需要哪些其他内容。最后，再来部署这个Java聊天应用程序。

12.1 填补空白：支持聊天室功能

如果读者认真阅读了上一章，那么就会发现，我在一些客户端的方法里造假了。在第10章中定义的RPC接口只有两种方法，但我使用了很多额外的方法。原始的RPC接口确实没有能使应用程序跑起来的全部内容。当开发者刚开始使用像App Engine这样的系统时，常常容易发生定义的接口不完全这样的问题。

分布式应用程序编程与传统的应用程序编程非常不同。做分布式应用编程，系统必须极其明确地分成客户端和服务器端两部分，如果开发者还不习惯用这种方式思考，就很容易忘记一些必需的基本东西。

设计原始的聊天界面时，我们将重点完全放在如何获得和发布聊天消息上。毕竟，这是一个聊天应用程序中的两项基本操作。

但是，聊天程序还有很多其他内容。聊天应用的用户的主要活动是发布和阅读消息。为了发布和阅读消息，用户需要能够看到哪些聊天室可用，并选择其中一个。我们需要全盘思考应用程序的整个生命周期，而不能只专注于一两个主要活动。不但需要考虑用户如何开展他们的活动，而且还需要考虑如何设置必要的基础设施，才能开展这些活动。

目前完成的聊天应用程序缺失了以下两项重要内容。

- 需要获得可用聊天室列表的方法，以使用户可以选择他们希望加入的聊天室。
- 需要创建聊天室的方法。在第一次将应用程序部署到服务器时，数据仓库完全是空的，不会有任何聊天室。为了生成一个可用的应用程序，我们需要设置一组聊天室来初始化数据仓库，或者为用户提供某种方法来定义新的聊天室。无论采用哪种方式，我们都必须提供创建聊天室的调用。

下面，让我们开始创建这些缺失的部分。

12.1.1 实现ChatRoom类

我们需要给接口添加几个新的RPC方法，但缺少数据类型。如前所述，我们需要能够创建和查询聊天室列表。要实现此功能，就必须有一个持久性的ChatRoom类。

ChatRoom对象应该是什么样子呢？为了能够创建和查询聊天室，并不需要实现得太过复杂，事实上，我们唯一真正需要的就是聊天室名称。

但在第一次定义该接口时，我们却搞砸了，漏掉了真正需要的东西。这次不要着急冒进，要仔细些，确保得到的ChatRoom类的设计完全正确。那么，我们可能希望在聊天室列表中有哪些内容呢？能想到的一个内容是时间戳，时间戳包含最后一条消息发送到该聊天室的时间。通过时间戳，我们可以在用户界面上展示给用户哪些聊天室是活跃的。因此，我们将编写一个持久性的聊天室对象，既包含聊天室的名称，也包含最后发布的消息的时间戳。这个类没有任何特殊的新内容。它是一个典型的持久性App Engine对象，因此，我们不再详细讨论。下面就是代码：

workspace/Chat/src/com/pragprog/aebook/chat/client/ChatRoom.java

```
public class ChatRoom implements IsSerializable {

    String name;
    long date;

    public ChatRoom(String chat, long date) {
        this.date = date;
        this.name = chat;
    }

    public ChatRoom() {
    }

    public String getName() {
        return name;
    }

    public long getLastMessageDate() {
        return date;
    }

    public void updateLastMessageDate(long d) {
        date = d;
    }
}
```

12.1.2 持久性的类和GWT

现在，我们遇到为数不多的GWT将Java自动转换成JavaScript的策略会引发问题的情况。我

们已经有了聊天消息类和聊天室类，既可以用在RPC调用中，也可以用在服务器的数据仓库的持久性代码中。问题是，如果用持久性的方式使用该类，该类就需要有一些持久性注解，而且需要一个持久性的关键字。但是，GWT不能将注解和关键字转化为JavaScript的内容！因此，我们不能对GWT的RPC使用持久性的类。

我们也不能使用这些类的GWT RPC版本进行持久化，因为RPC类不具备数据仓库JDO实现所需要的注解和字段。因此，我们需要为每个类实现两个版本，一个用于持久化，一个用于RPC。这很烦人，但这是解决这一痛苦问题的最简单的方式。

我们把`ChatMessage`和`ChatRoom`类的RPC版本放在`client`软件包中，把它们的持久性版本放在`server`软件包中。为了使代码变得更容易阅读，我们在类的持久性版本名字前使用前缀*P*，稍后在对这两个不同版本进行相互转化时，就会明白其中原因。

现在，为了使读者可以看出其中的差别，来看看新的`ChatRoom`的持久性版本。我们刚刚已经看过了它的RPC版本，下面是数据仓库中的持久性版本：

workspace/Chat/src/com/pragprog/aebook/chat/server/PChatRoom.java

```
@PersistenceCapable(identityType = IdentityType.APPLICATION)
public class PChatRoom {
    @PrimaryKey
    @Persistent(valueStrategy = IdGeneratorStrategy.IDENTITY)
    private Key key;

    @Persistent
    String name;

    @Persistent
    long date;

    public PChatRoom() {
    }

    public PChatRoom(String chat, long date) {
        this.date = date;
        this.name = chat;
    }

    public ChatRoom asChatRoom() {
        return new ChatRoom(name, date);
    }

    public String getName() {
        return name;
    }

    public Key getKey() {
        return key;
    }

    public long getLastMessageDate() {
```

```
            return date;
        }

        public void updateLastMessageDate(long d) {
            date = d;
        }
    }
```


12.1.3 服务器端的ChatRoom方法

现在，我们已经完成ChatRoom类，可以编写使用它的方法了。我们到底希望能够使用ChatRoom做什么呢？

- **创建聊天室**

  需要能够创建聊天室。

- **显示聊天室列表**

  需要能够得到一个可用的聊天室列表。

- **删除聊天室**

  这是可选的，取决于我们希望系统如何工作。如果认为聊天室本质上是瞬间的——也就是说，不断地创建，然后丢弃——那么我们希望，一旦聊天室不再使用时，有一种方式可以清除并且丢弃聊天室。另一方面，如果希望聊天室是一个用户持续交谈的永久记录，那么就不应该删除聊天室。

  我更加倾向于后面这种聊天室使用方式。大多数聊天室都允许用户持续交谈，我可能今天会不再谈话，但很有可能到明天想说些什么时，会回来继续说。所以，我决定不考虑删除这个功能。

知道了所要的方法后，我们就可以将这些方法添加到ChatService。新方法如下所示：

workspace/Chat/src/com/pragprog/aebook/chat/client/ChatService.java

```
    List<ChatRoom> getChats();

    void addChat(String chatname);
```

正如一直在GWT中所做的一样，我们需要给该接口的异步版本添加相应的方法：

workspace/Chat/src/com/pragprog/aebook/chat/ client/ChatServiceAsync.java

```
        void getChats(AsyncCallback<List<ChatRoom>> chats);

    void addChat(String chatname,
            AsyncCallback<Void> callback);
```

这些实现都大同小异。getChats几乎是最简单可行的JDOQL查询，它获取所有ChatRoom类型的对象并将其返回，如下所示：

workspace/Chat/src/com/pragprog/aebook/chat/server/ChatServiceImpl.java

```
@SuppressWarnings("unchecked")
public List<ChatRoom> getChats() {
    PersistenceManager persister = Persister.getPersistenceManager();
    try {
        Query query = persister.newQuery(ChatRoom.class);
        query.setOrdering("date");
        return (List<ChatRoom>)query.execute();
    } finally {
        persister.close();
    }
}
```

添加新聊天室会稍微复杂一些。我们需要建立一个聊天室对象，并使其具有持久性：

workspace/Chat/src/com/pragprog/aebook/chat/server/ChatServiceImpl.java

```
public void addChat(String chat) {
    PersistenceManager persister = Persister.getPersistenceManager();
    try {
        PChatRoom newchat =
            new PChatRoom(chat, System.currentTimeMillis());
        persister.makePersistent(newchat);
    } finally {
        persister.close();
    }
}
```

12.2　适当的交互式设计：增量式设计

原来的接口还有另一个大问题。我们只提供了一个消息，用来在指定的聊天室中获取所有的消息。构建的用户界面中，聊天室从不会被删除，因此，消息只会不断地累积。同时，我们希望应用程序是交互式的，这也就意味着，它会不断地更新消息的显示列表。当我们考虑这两点后，就会发现它既浪费资源，又损失性能，因为必须不断地重新发送整个消息列表。即使假设我们只在发布新消息的时候取回消息列表，也意味着在客户端，我们第一次发布消息时要取回消息一，第二次要取回消息一和消息二，第三次是消息一、二、三，然后是一、二、三、四……依此类推，到第十条消息时，我们已经将消息一发送了十次。回想一下，系统会每秒自动更新两次，也就是说，在使用系统的一分钟内，我们将获取相同的消息120次。

这可不是云应用程序应有的工作方式。假想一下，应用程序并不会只有一个用户，它有上百、上千，或者更多用户！假设有一千个用户时，我们每秒会将同一个旧消息重新发送12万次，这可不仅是浪费时间，而且也是浪费钱！在App Engine中，我们要为所用的资源付费。如果每一秒钟都将同一内容重新发送10万次，免费资源将会很快用尽，然后就该为所用的带宽付费了。

绝不能这样做。在真正的云应用程序中，我们并不是非常担心CPU时间，CPU时间很便宜。我们关注的是通信，因为通信的时间和金钱都很昂贵。与计算这些内容所耗费的时间相比，通过网络发送内容简直慢得难以想象。绝大多数时候，设计工作的重点应在于如何最大限度地降低通

信。多次重新计算相同的内容比通过网络发送一次内容更快、更便宜。

对于这个聊天室范例来说，没必要给单一的客户端重新发送消息。客户端可以记住它已经看到的消息，从而只需要将新消息添加到其列表中即可。在这个程序的云实现中，可以基于时间取回列表进行更新——也就是说，我们可以告诉服务器端："给我上次查询之后的所有聊天消息。"

这种工作方式称为增量式（incremental）更新。我们并不是重新获取整个数据集合，而是在客户端和服务器端同时管理数据的副本，并只发送很小的修改。对于几乎所有云应用程序而言，如果希望它操作高效且负载得起，那么我们就需要将其设计为增量式更新。

12.2.1 增量式更新的数据对象

要实现增量式更新，通常需要建立一个数据结构，从而浑然一体地容纳增量式。在我们的例子中，这意味着不能只从getMessages或getMessagesSince方法返回一个ChatMessages列表，还需要返回消息列表和时间戳。因此需要创建一个新的对象类型，封装这两个内容。该对象不必是一个持久性对象，因为永远不会将其存储在数据仓库中，它只需动态生成以响应客户端请求即可。但它确实需要通过网络进行发送，因此，它需要被序列化（serializable），这也就意味着GWT生成的代码既可将对象转换为可以在消息中发送的格式，也可以将消息转换为对象。在GWT中，我们通过为对象实现接口IsSerializable将其序列化。该接口没有方法，它只是一个标识，用来告诉GWT需要生成序列化该对象的代码。[①]

了解上述内容之后，写一个序列化对象非常容易，只要使其声明并实现IsSerializable，然后确保它包含一个默认的无参数的构造函数就可以了。因此，我们的序列化对象简单地封装了ChatMessages列表和一个将用作时间戳的日期对象。我们希望GWT对此进行转换。实现该功能的最简单的办法，就是把它放在client软件包中。

workspace/Chat/src/com/pragprog/aebook/chat/client/ChatMessageList.java

```
public class ChatMessageList implements IsSerializable {

    private List<ChatMessage> messages;
    private long time;
    private String chat;

    public ChatMessageList(String chat, long time) {
      this.chat = chat;
      this.time = time;
      this.messages = new ArrayList<ChatMessage>();
    }
```

① 我们实际上可以为此使用标准的Java接口Serializable。但是，Serializable被Java虚拟机用来识别哪些是可以使用本地Java序列化机制序列化的内容。GWT不使用Java序列化——其实，GWT序列化甚至不重构标准的Java序列化。因此，我倾向于使用GWT自身的标识接口，使其清晰地表明我在使用GWT序列化。

```
    /**
     * GWT序列化的默认零参数构造函数
     */
    public ChatMessageList() {
        messages = new ArrayList<ChatMessage>();
        time = System.currentTimeMillis();
        chat = null;
    }

    public String getChat() {
        return chat;
    }

    public List<ChatMessage> getMessages() {
        return messages;
    }

    public long getTimestamp() {
        return time;
    }

    public void addMessage(ChatMessage msg) {
        messages.add(msg);
    }

    public void addMessages(List<ChatMessage> messages) {
        messages.addAll(messages);
    }
}
```

12.2.2 增量式的聊天室界面

为了使用这种基于时间的增量式取回机制，我们需要修改一些旧方法，并且在界面中添加一些新方法：

workspace/Chat/src/com/pragprog/aebook/chat/client/ChatService.java

```
❶  void postMessage(ChatMessage messages);
❷  ChatMessageList getMessages(String room);
❸  ChatMessageList getMessagesSince(String chat, long timestamp);
```

❶ 最初的PostMessage方法是返回一个聊天室的消息列表。但是，我们想要添加一个方法来获得特定时间点后的消息（即，不想一遍又一遍地发送相同的旧消息），适用于PostMessage方法的返回结果，也适用于getMessages的返回结果。我们有两种选择：可以给PostMessage添加一个时间戳参数，或者可以使PostMessage不返回值，相反，使客户端在新消息发布后调用getMessages。

作为一般性的规则，最好是将查询方法（取回值的方法）和更新方法（修改值的方法）分离开。我们将遵循这一规则，使PostMessage的返回值为空。

❷ 当用户第一次连接到聊天室时，客户端应该获取该聊天室的所有消息。之后，它将开始做增量式获取。为了使这种方式能够工作，客户端需要知道它们第一次获取操作的服务器时间。因此，我们必须修改getMessages调用的返回类型，使它返回一个ChatMessageList。

❸ 现在，我们终于有了一个新方法：getMessagesSince。

像以往针对GWT的实现一样，我们需要提供一个异步版本：

workspace/Chat/src/com/pragprog/aebook/chat/client/ChatServiceAsync.java

```
void postMessage(ChatMessage message,
                 AsyncCallback<Void> callback);

    void getMessages(String chatroom,
                     AsyncCallback<ChatMessageList> callback);

    void getMessagesSince(String chat, long timestamp,
                     AsyncCallback<ChatMessageList> callback);
```

12.2.3 解决时间难题

在云中，基于时间的工作确实需要用点技巧。客户端不是在同一台服务器上运行。事实上，服务器并不是只有一个，因为服务器端代码可能会在云中许多不同的机器上运行，所以无法保证客户端所使用的时钟和服务器端是同步的。也就是说，客户端上次取回消息的时间可能与服务器端认为客户端取回消息的时间不同。更糟的是，网络还可能会出现各种问题，从而导致延时增加，使得基于时间的工作更加复杂。这是一个潜在的严重问题——幸运的是，只要开发者清楚问题的关键，也就不太难解决。首先让我们分析一下问题。

假设客户端和服务器端的时钟完全同步。由于网络延迟所造成的简单计时问题，我们仍然可能遇到冲突。想象一下这样的情景：

09:34:58.1432 客户端发送消息请求；

09:23:58.1894 由于网络传输时间，服务器端在略微延时后收到该请求；

09:23:59.2401 服务器端处理请求，并进行响应；

09:24:59.4019 客户端通过网络接收响应。

客户端应该使用哪个时间作为其请求的时间呢？假设客户端用时间58.1432作为发送请求的时间。另一个客户端可能已经在58.1434发送了消息X。当服务器端收到请求时，会返回一个包含X的列表。下一次客户端请求消息时，它会请求比58.1432更新的消息。由于X的时间戳是58.1434，它比58.1432新，因此服务器端将再次在其响应中包括消息X。这意味着第一个客户端将看到消息X两次。这显然不是我们想要的。

我们还可以使用客户端收到响应的时间：59.4019。也许另一个客户端在59.2530发送消息Y。那么，消息Y将不会出现在客户端收到的消息列表中，因为它是在服务器端给客户端发送响应后才被发布的。客户端的下一个请求将针对发布在59.4019后的消息，因为Y的时间戳是59.2530，这意味着它将不被包括在内。消息Y将永远不会被发送到客户端。同样，这显然也不是我们想要的。

看起来不管我们选择哪个时间，都会丢失消息。在一般情况下，解决的办法是使用单一时钟。在云应用程序中，任何依赖于两个不同时钟的内容，都是有问题的，必须始终以一个时钟工作。我们不能以客户端的时钟工作，因为客户端的时钟不在我们的可控范围之内：会有多个客户端，并且没办法知道它们是否同步。但是，我们可以依赖App Engine服务器中的时钟，它们使用网络时间协议同步。可以肯定的是，它们之间的时钟差异比我们可以测量的时间单元要更小。

服务器时钟，即服务器代码中由App Engine提供的时钟，才是我们应该使用的唯一时钟。这意味着当请求被送达时，需要告诉客户端服务器上是什么时间。

12.2.4　实现服务器端的方法

既然已经制定了客户端和服务器端之间的接口，接下来就来实现服务器端代码。之前已经实现了其中的某些部分，但我们已经改变了其内容，并增加了更多的方法，因此在服务器端，我们几乎要从零开始。

可以从getMessages开始入手。其代码几乎和以前一样，只是现在需要把结果封装在带有时间戳的MessageList对象里。

workspace/Chat/src/com/pragprog/aebook/chat/server/ChatServiceImpl.java

```
@SuppressWarnings("unchecked")
public ChatMessageList getMessages(String chat) {
    PersistenceManager persister = Persister.getPersistenceManager();
    try {
        Query query = persister.newQuery(PChatMessage.class);
        query.setFilter("chat == desiredRoom");
        query.declareParameters("String desiredRoom");
        query.setOrdering("date");
        List<PChatMessage> messages = (List<PChatMessage>)query.execute(chat);
        // 获取最新消息
        ChatMessageList result = null;
        if (messages.size() > 1) {
❶            PChatMessage lastMessage = messages.get(messages.size() - 1);
            result = new ChatMessageList(chat, lastMessage.getDate());
            for (PChatMessage pchatmsg : messages) {
            result.addMessage(pchatmsg.asChatMessage());
            }
        } else {
            result = new ChatMessageList(chat, System.currentTimeMillis());
        }
        return result;
    } finally {
        persister.close();
    }
}
```

这段代码中有一个有趣的片段，与12.2.3的内容有关。在1处，我们得到了查询语句返回的最后一条消息，并使用其时间戳作为时间。我们这样做，而不采用系统时间的原因是，应用程序不是在一台服务器上运行的。但是，有可能云中的一台服务器在处理一个POST请求的同时，也在处理

一个GET请求，而且可能在查询语句结束时和getMessages方法实现时的时间之间引发新的消息发布。因此，可能有一个消息的时间戳是在上一条取回的消息和getMessages记录的当前时间之间。为了避免这种情形，我们只采用上一条消息的时间。该策略为我们提供了一致的时间视角，以便增量式消息取回可以正常工作。

接下来，我们可以看看如何发布一条消息。从这里开始，事情开始真的变得有趣了。

workspace/Chat/src/com/pragprog/aebook/chat/server/ChatServiceImpl.java

```
    @SuppressWarnings("unchecked")
public void postMessage(ChatMessage message) {
        UserService userService = UserServiceFactory.getUserService();
        User user = userService.getCurrentUser();
    PersistenceManager persister = Persister.getPersistenceManager();
        try {
        PChatMessage pmessage = new PChatMessage(user.getNickname(),
                                        message.getMessage(),
                                        message.getChat());
❶       long timestamp = System.currentTimeMillis();
        pmessage.setDate(timestamp);
        persister.makePersistent(pmessage);

❷       Query query = persister.newQuery(PChatRoom.class);
        query.setFilter("name == " + message.getChat());
        List<PChatRoom> chats = (List<PChatRoom>) query.execute();
❸       PChatRoom chat = chats.get(0);
❹       chat.updateLastMessageDate(timestamp);
    } finally {
        persister.close();
    }
}
```

我们从App Engine的一些典型样板开始：创建一个PersistenceManager，使聊天信息对象持久化，从而使该对象能够在数据仓库中存储。然后来看看新内容，如下所述。

❶ 修改消息的日期。正如我前面所解释的，必须非常谨慎，只使用一个时钟，也就是App Engine的时钟。我们不考虑客户端的日期和时间，需要用的是服务器上的日期和时间。

❷ Chat对象必须包含上一条发布消息的时间戳。因此，我们需要取回聊天对象，将其上一条消息的时间更新为该消息的时间。

❸ 查询语句返回聊天室列表，但我们知道，该列表只有一个条目，因此，从列表中获取这个唯一条目。

❹ 现在更新聊天对象的上一条消息的日期。不需要保存该消息，因为会使用PersistenceManager查询语句取回该消息，所以，它已经由PersistenceManager管理。这意味着消息的更新在事务结束时会被自动保存。

最后，我们需要增量式的getMessagesSince：

workspace/Chat/src/com/pragprog/aebook/chat/server/ChatServiceImpl.java

```
@SuppressWarnings("unchecked")
public ChatMessageList getMessagesSince(String chat, long timestamp) {
    PersistenceManager persister = Persister.getPersistenceManager();
    try {
        Query query = persister.newQuery(PChatMessage.class);
❶       query.declareParameters("String desiredRoom, int earliest");
❷       query.setFilter("chat == desiredRoom && date > earliest");
        query.setOrdering("date");
        List<PChatMessage> messages =
            (List<PChatMessage>)query.execute(chat, timestamp);
        ChatMessageList msgList = null;
        // 获取最新消息
        if (messages.size() >= 1) {
            PChatMessage lastMessage = messages.get(messages.size() - 1);
            msgList = new ChatMessageList(chat, lastMessage.getDate());
        } else {
            msgList = new ChatMessageList(chat, System.currentTimeMillis());
        }
        for (PChatMessage msg : messages) {
            msgList.addMessage(msg.asChatMessage());
        }
        return msgList;
    } finally {
        persister.close();
    }
}
```

这个方法几乎与更新的`getMessages`一样，除了JDOQL查询语句的以下两个改变。

❶ 我们给查询添加了一个新的参数，从而可以比较传递给调用的日期和`getMessagesSince`得到的消息的日期。

❷ 给过滤器添加了一个新的子句以比较日期。

12.3　更新客户端

现在终于有了与客户端交互的所有服务器端的方法！应用程序也已经非常接近可运行状态。但仍然需要对客户端进行一些修改，使其知道如何使用更新的RPC接口。

这里要做的事情并不多，基本上，唯一要处理的问题是，更新代码以适应RPC服务里所作的修改。客户端不再获取聊天消息列表，现在获取的是`ChatMessageList`，而且现在每当收到`ChatMessageList`时，都需要更新其存储的上一条消息的时间，以便客户端可以使用该时间来进行增量式更新请求。我们需要做的全部工作就是更新`addNewMessages`方法：

workspace/Chat/src/com/pragprog/aebook/chat/client/Chat.java

```
protected void addNewMessages(ChatMessageList newMessages) {
    lastMessageTime = newMessages.getTimestamp();
    StringBuilder content = new StringBuilder();
    content.append(text.getText());
```

```
        for (ChatMessage cm : newMessages.getMessages()) {
            content.append(renderChatMessage(cm));
        }
        text.setText(content.toString());
    }
```

然后，我们还需要更新创建了定时更新的代码，以便它在其增量式更新请求中使用lastMessageTime：

```
// 创建一个新定时器
Timer elapsedTimer = new Timer() {
    public void run() {
        chatService.getMessagesSince(getCurrentChat(), lastMessageTime,
                new MessageListCallback());
    }
};
// 每500毫秒更新定时器
elapsedTimer.scheduleRepeating(500);
```


12.4　聊天室管理

在可以运行我们的聊天应用程序之前，还有最后一件事情要做。我们提供了一个在服务器上创建聊天室的方法，但还没有为其实现任何接口。就目前而言，我们并不打算建立一个新的用户界面元素用于添加聊天室，而是将编写一些*管理代码*（Administration Code）。管理代码是开发者编写的用以实现设置、清除、初始化或监视的代码。管理代码不是供用户访问的代码，而是供我们用来管理系统的。

很多时候，开发者会为自己的管理代码构建一个用户界面。例如，如果开发者想要查看有多少人访问聊天系统，有多少消息发布到哪个聊天室，以及人们访问该系统的频率，那么很可能会想要一个用户界面。然后，他们会建立另一个GWT用户界面来做管理。为了管理各项工作，开发者可能只会在自己的浏览器中加载管理者用户界面的URL。如果只有开发者去加载管理者用户界面，那么，需要仔细地提供一些安全机制（后面将会详细介绍），确保只有开发者自己才可以访问管理用户界面。当然，还有其他种类的管理代码也很有价值。例如第一次部署一台服务器时，需要初始化一些内容。聊天应用程序需要初始化一组聊天室。

问题在于，并非只在服务器上运行代码就能完成设置。我们受限于App Engine所提供的接口，需要做*自我检测初始化*（self-detecting initialization），也就是说，需要插入代码以检测服务器端是否已被初始化，如果没有，则调用初始化方法。

可以通过修改getChats实现初始化功能。每次用户进入应用程序且在看到任何内容之前，聊天应用程序会调用getChats来初始化聊天室列表视图。所以要做的工作就很简单了。当应用程序取回聊天室列表时，我们将进行检查，以查看列表是否为空。如果是，则调用一个方法初始化一组聊天室。我们将初始化方法放在服务器的实现中，但不会将其声明为RPC。我们不希望用户能够调用该初始化方法，它被调用的唯一方式应该就是当系统检测到聊天室还未被初始化时自动调用。

初始化聊天室的代码非常简单。我们创建一个`PersistenceManager`，创建一些聊天室，并将它们持久化：

workspace/Chat/src/com/pragprog/aebook/chat/server/ChatServiceImpl.java

```
static final String[] DEFAULT_ROOMS =
    new String[] { "chat", "book", "java", "python" };

public List<ChatRoom> initializeChats(PersistenceManager persister) {
    List<ChatRoom> rooms = new ArrayList<ChatRoom>();
    List<PChatRoom> prooms = new ArrayList<PChatRoom>();
    long now = System.currentTimeMillis();
    for (String name : DEFAULT_ROOMS) {
        PChatRoom r = new PChatRoom(name, now);
        prooms.add(r);
        rooms.add(r.asChatRoom());
        persister.makePersistent(r);
    }
    return rooms;
}
```

为了调用该方法，我们只需给`getChats`增加一个测试，检查一下数据仓库中的聊天室列表是否为空。如果是，我们调用`initializeChats`：

workspace/Chat/src/com/pragprog/aebook/chat/server/ChatServiceImpl.java

```
@SuppressWarnings("unchecked")
public List<ChatRoom> getChats() {
    PersistenceManager persister = Persister.getPersistenceManager();
    try {
        Query query = persister.newQuery(PChatRoom.class);
        query.setOrdering("date");
        List<PChatRoom> rooms = (List<PChatRoom>)query.execute();
        if (rooms.isEmpty()) {
            return initializeChats(persister);
        } else {
            List<ChatRoom> result = new ArrayList<ChatRoom>();
            for (PChatRoom pchatroom : rooms) {
                result.add(pchatroom.asChatRoom());
            }
            return result;
        }
    }
    finally {
        persister.close();
    }
}
```

12.5　运行和部署聊天应用程序

我们终于完成了Java版本的聊天室！现在该测试它了。在Eclipse中，开发者只要使用“运行”（Run）按钮就可以运行其应用程序。应用程序启动需要几分钟，然后会告诉开发者一个用来访问测试服务器的URL。开发者访问该URL时，就会被要求安装一个浏览器插件：为了测试目的，它使用一种特殊的机制来管理客户端和服务器端之间的通信，以便开发者可以使用Eclipse调试器跟踪客户端和服务器端两端的工作。（事实上，开发者甚至可以跟踪RPC，从RPC在客户端被调用的地方到其真正在服务器上被执行。）Eclipse使App Engine程序的调试与传统的应用程序调试几乎一样简单。

在开发者可以将其Java代码部署到App Engine服务器之前，需要进行一个完整的GWT编译。还是靠Eclipse使事情变得很简单。首先，开发者需要进行一个全面GWT编译。通常情况下，Eclipse做的是部分编译，基本上只使用受限模式的Java编译器。然而，为了部署应用程序，开发者需要进行完整编译，实现Java到JavaScript的完全转换。Eclipse工具栏的上方，有一个看起来像红色工具箱的按钮，标有Google的“G:”，只要按一下该按钮，Eclipse就会做全面GWT编译。

现在，我们的应用程序终于可以运行了！紧邻着刚才用来编译应用程序的工具栏按钮，有一个看起来像App Engine喷气发动机标志的按钮。请点击该按钮。开发者第一次点击该按钮时，它会提示开发者输入App Engine应用程序的ID，然后它会进行部署。做一次完整部署需要花费几分钟，但是，一旦完成，开发者的应用程序就在App Engine中“居住”下来了。

那么，完成所有这些工作之后，应用程序看起来是什么样子呢？请将视线移动到下面的图12-1上。



图12-1　已部署的Java版聊天室

很漂亮，是吧？

就像Python应用程序一样，开发者现在拥有应用程序控制面板上所有管理控制的访问权。

故障排除

当开发者部署应用程序时，有几个问题经常发生，但是，其错误消息看起来怪异和神秘，并

且，在最坏的情况下，极具误导性。这里有两个最常见的怪异错误，如下所述。

- 当开发者改变在数据仓库存储数据的方式时，经常会得到各种奇怪的错误消息，例如，类强制转换异常和“无法转换数据类型”。这两个错误信息是由于开发者的数据仓库包含了修改存储方式之前的数据所造成的，导致了数据仓库中的数据不一致。因此，开发者最好在部署应用程序之前确定好数据结构。但是，在工作过程中，当确定数据结构时，开发者会经常需要改变内容。例如，当我调整本章的代码时，最初将数据存储为一个`java.util.Date`的对象，但后来发现该对象竟然引发了一个GWT的RPC问题，所以我将时间戳的表示改为`long`型，用来指明从某个时间点开始的、以毫秒计的时间戳。

  这些错误并不是由开发者代码中的任何问题引起的，完全是由留在开发者的本地数据仓库的旧数据造成的。要修正这样的错误，需要做的就是请空本地数据仓库的旧数据。为此，开发者只需要删除其Java项目中`WEB-INF/appengine-generated/local_db.bin`目录中的内容。如果已经实际部署了应用程序，那么就需要去应用程序的仪表板，选择“数据仓库视图”（Datastore Viewer），在那里，可以进行“全部选择”（Select All），然后删除所有需要清除的对象。
- Java App Engine最神秘的，但又最常见的错误之一出现在GWT RPC参数中。当开发者运行其应用程序时，会得到错误消息：“类型（type）不包括在能够被此序列化策略（Serialization-Policy）序列化的类型组中。”这似乎是暗示配置有问题，但该错误非常具有误导性。发生错误的原因只是开发者在该类型中遗漏了一个零参数的构造函数。GWT使用的每一个序列化的类型都必须有一个公共的、零参数的构造方法。它并不需要做任何实际的初始化，它只是被GWT基础架构用来创建一个空白的对象，然后GWT会用反序列化消息的结果填充该对象。

12.6　服务器端结束语

过去的几章介绍了大量内容。我们已经看到了如何在Java中实现数据仓库持久性，学到了很多GWT的有关知识，了解了如何使用GWT的模型设计应用程序。关于如何构建一个基于RPC的客户端和服务器端之间的接口，以及如何使用该接口生成性能良好的网络应用程序方面，我们做了很多工作。

从这里开始，将要切换到更高级的主题。我们不再特别专注于Java或Python，而是去了解一些不同的主题：安全和认证、高级数据管理、管理和实施，以及监控。对于每一个主题，我们都将会讲解如何在Java和Python中实现。

Part 4

第四部分

高级 Google App Engine 编程

本部分内容

第 13 章

高级数据仓库：特性类型

我们在学习Google App Engine编程时，已经使用数据仓库完成了一些基本的工作。虽然我们没有做什么特别困难的事情，但是，即使是在已经实现的简单的应用程序中，数据仓库也是我们设计的核心。当开发人员开始构建更加复杂的App Engine程序时，就会发现，数据仓库变得更加重要了。

数据仓库能够做的事情远远多于我们到目前为止所看到的功能。它是一个非常强大、非常灵活的持久性存储系统。在本章中，我们将了解数据仓库有哪些能力，并试验它的一些高级功能。

从本章开始将采取一种不同的方法。我们不会把自己局限于Java或Python，而是要同时考虑二者。我们也不去构建用户界面，而是将重点放在数据管理。我们将会看到，可以使用HTTP为App Engine应用程序构建服务，该方式允许我们使用任意最便捷的编程语言实现每个服务。我们可以使用HTTP层作为编程语言之间的桥梁，从而能够编写使用Java服务的Python服务，反之亦然。

13.1 构建文件系统服务

纵观高级主题部分，我们不再继续把重点放在聊天应用程序上，而将建立不同种类的服务，以展示Google App Engine的各个部分。在这一章中，我们将使用数据仓库构建一个类似文件系统的服务。

我们可以认为这是聊天服务的附加功能的基础设施。许多聊天系统（例如Google Talk）为用户提供共享和交换文件的能力。我们的文件系统服务可以用来实现聊天应用程序中一个类似的文件系统。

在开始在Google App Engine中实现文件系统服务之前，先需要弄清楚我们需要的是什么。由于这是一个基于网络的文件系统，所以，它的行为不会和传统操作系统的本地磁盘文件系统完全一样。

基于网络的文件系统应该是什么样子的呢？实际上已经有一个实现网络文件系统的标准方式了，基于一个称为WebDAV的HTTP扩展版本。WebDAV实在太复杂了，难以作为例子实现，但是我们可以将其看作一个粗略的模型。在WebDAV中，每一个URL标识了一个资源，资源基本上是一个文件。每个资源有内容，内容是字节、特性的集合，特性是从名称到元数据块的映射。由于数据仓库使用property（特性）作为其模型的字段名称，我们将使用attribute（属性）代替。

REST 风格编程

在本章中，我们将围绕 REST 模型来构建应用程序，我们将要做的编程过程通常被称为 REST 风格编程。REST 是 Representational State transfer（表示状态转移）的简称，其含义是：开发者所编写的程序使用基于网络的协议时就是按照网络协议原先预期的方式访问和更新网络上的资源的。

REST 已成为一个时髦词——每个人都希望声称他们的系统是 REST 风格的，并且其技术是实现 REST 的最好的方式。然而，REST 实际上并不复杂。

REST 的含义是，开发者使用基本的 HTTP 原语（GET、PUT 和 POST）时是完全按照它们原定的方式使用的。如果要取回数据，你总是使用 GET；要存储一个完整的对象，你会使用 PUT；要更新数据，你会使用 POST。通常，在 REST 中，开发者不为 GET 或 PUT 指定类似于 CGI 参数的内容：因为一个 URL 标识了一个特定的资源，而且由于 GET 意味着"取回资源"，所以，开发者取回资源时只需用到 URL，应该不需要更多的其他内容。

由于 REST 的简洁性，它很强大。很多基于网络的编程都是以人难以置信的草率的方式实现的。有大量的网络和云应用程序在 GET 中使用带 CGI 参数的长长的字符串，以完成除了取回数据之外的各种事情。在许多方面，REST 就像网络中的面向对象：它是一种编程的方式，围绕着开发者正在使用的基本实体和这些实体需要的基本操作进行构建。

因此，例如，在传统的本地文件系统中，用户会有元数据，如所有者、创建时间、访问权限等。而在我们的类似于WebDAV的文件系统中，这些都将使用属性来处理。

有一件起初有点令人意外的事情是，在WebDAV中，目录只是一个普通的资源。一个目录资源和一个非目录资源之间的唯一区别在于目录有一个属性，该属性包含了子资源名称列表。

下面来实现该文件系统。首先，我们只是写一些标准的Python代码，然后逐步把Python代码转换为数据仓库版本。因此，我们将从用Python编写的一个非常基本的伪文件系统的代码框架开始。这并不是Google Apple Engine代码，它只是标准的Python代码，用来表述我们所希望的系统执行方式。根据我的经验，这往往是构建Google Apple Engine应用程序或服务中非常有价值的第一步。这一步意味着开发者构建云应用程序时，已经弄清楚了希望系统如何执行的一系列传统问题，也理解了该程序在基于云的环境中执行的各种新问题。做好第一个框架，有助于开发者在开始云编程之前，理清基本的设计问题。

filesystem/filesystem.py

```
from datetime import datetime
import string

class Resource(object):
    @staticmethod
    def MakeResource():
        return Resource(content=None, attributes={})

    def fs_put(self, content):
        self.content = content

    def fs_get(self):
        return self.content

    def fs_setAttribute(self, name, value):
        self.attributes[name] = value

    def fs_getAttribute(self, name):
        return self.attributes[name]

    def isDir(self):
        return self.getAttribute("children") is not None

    def addChild(self, name, resource):
        if self.getAttribute("children") is None:
            self.setAttribute("children", {})
        self.getAttribute("children")[name] = resource

class FileSystem(object):
    @staticmethod
    def MakeFilesystem():
        fs = FileSystem()
        fs.root = FileSystem.MakeFile("/", "root", "")
        return self

    @staticmethod
    def MakeFile(name, owner, content):
        file = Resource()
        file.put(content)
        file.setAttribute("owner", owner)
        file.setAttribute("time", datetime.now())
        return file

    def getRoot(self):
        return self.root

    def getResourceFromChild(self, child, nameElements):
        """获取文件路径的一个递归处理函数。child是这个递归调用目标中的目录名。nameElements则是child随后路径
名的各种组成部分。每次递归调用都会取出一部分路径，然后在余下的路径中再调用自身。当nameElements为空时，也就获取
到了整个路径。"""
        if nameElements is []:
            return child
```

```
        childsChildren = child.getAttribute("children")
        if childsChildren is None:
            return None
        else:
            nextChild = childsChildren[nameElements[0]]
            if nextChild is None:
                return None
            else:
                return self.getResourceFromChild(nextChild, nameElements[1:])

    def getResourceAtPath(self, path):
        pathElements = string.split(path, "/")
        self.getResourceFromChild(self.getRoot(), pathElements)
```


上面的代码非常简单，所以我并不打算深入解释。文件系统就是一个对象，有一个根目录。目录仅仅是一个资源，有一个名为`children`的属性，该属性是从名称到资源的映射。文件系统要实现解析复杂路径得到资源的方法。

13.2　浅尝文件系统建模

看一下上边的非数据仓库版本的文件系统，让我们想想如何将其转化成数据仓库的持久性版本。该文件系统确实简单，它是一个持久性对象，具有一个特性——根资源。文件系统对象最终将成为一个与servlet相关的对象，所有访问该文件系统中的任何内容的调用都将通过与该文件系统对象相关联的单一的servlet处理。但现在，我们不去担心这一部分内容，只是创建一个文件系统模型。我们暂不实现各个方法（如`getResourceFromChild`）。在实现方法之前，首先需要理解我们的文件系统是如何表示的。

filesystem/persistent_filesystem.py

```
class FileSystem(object):
    def __init__(self):
        self.root = MakeFile("/", "root", "")
        return self
    def getRoot(self):
        return self.root

    def getResourceFromChild(self, child, nameElements):
        if nameElements is []:
            return child
        childsChildren = child.getProperty("children")
        if childsChildren is None:
            return None
        else:
            nextChild = childsChildren[nameElements[0]]
            if nextChild is None:
                return None
            else:
                return getResourceFromChild(nextChild, nameElements[1:])
```

```
    def getResourceAtPath(self, path):
        pathElements = string.split(path, "/")
```

资源也更为有趣。资源有两个特性：内容和属性。很明显，内容是一个大文件——一个巨大的字节组。我们不知道，也不关心除此之外内容是否有结构体。

但是，如何表示资源的属性是个麻烦。属性是从名称到值的映射，而值可以是任何东西！到目前为止，我们已经看到的Python持久性需要为持久化特性声明一个类型，但是，除了在原始Python中处理属性的方式之外，没有一种单一类型的值可以被所有属性使用。因此，我们在数据仓库中不能采用这种方式。

有两种方法可以解决这个问题。或者用一种方法将属性描述为能在数据仓库中声明的可储存对象，或者找到一些更灵活地存储东西的方式。现在，我们将尝试前者，要求所有的属性值必须是字符串。如果用户要使用更复杂的类型作为属性值，可以，但是当用户要在资源对象中存储该属性时，需要将其转换成字符串形式。这样做并不会过于繁琐，因为要在消息中使用这些内容，用户必须通过某种方式将对象转换为wire格式，如XML或JSON等，而且可以只使用这种表示方式。

当然，这并没有完全解决我们的问题。用目前已经看到的数据仓库模型，我们只能使用原子特性值——也就是说，不能存储列表或映射。

幸运的是，这不是一个真正的数据仓库的限制，只是目前为止看到的功能设置的限制。数据仓库不支持映射，但它支持列表，虽然得在感兴趣的值的列表部分做一些工作。后边会看到，在查询语句的帮助下，数据仓库能够做得足够好。

但现在，因为不能使用映射，所以我们将使用列表来构建映射。数据仓库的模型支持列表。列表有一些限制，但是可以完成这项工作。列表包括的值要具有相同的类型。而不管是Python原始对象还是数据仓库关键字都需要这些值。

在数据仓库中，每个存储的对象都有一个唯一的关键字。给定该关键字，用户可以取回和更新相应的对象。因此，对于属性，我们将使用一个关键字/值的二元组列表。关键字和值都是字符串。因此，要取回资源的一个属性，首先搜索资源的属性列表，如果找到所需的属性，则返回属性值。

综上，我们需要做的第一件事情是创建一个属性类型：

filesystem/first-persistent.py

```
class ResourceAttribute(db.Model):
    name = db.StringProperty(required=True)
    value = db.TextProperty(required=True)
```

属性是一种有着以下两个字段的对象。

- name

  字符串特性。字符串不能超过500个字符，但是，我们可以编写使用字符串值的查询语句，并且可以按照字符串特性对查询结果排序。我们可能希望能够做类似于“所有具有children特性的对象”的查询，从而取回所有目录，因此，需要使用字符串特性来表示name字段。

- value

  对于值，我们将使用文本特性。文本虽然可以是我们想要的任意长度，但是使用文本的方式比字符串更为受限。这里要做一个权衡。一方面，我们可能希望能够做一些工作，如编写获得特定用户创建的所有资源的查询，而要实现这样的查询，需要能够在查询中使用特性的值。但另一方面，我们不知道用户可能想要在特性中存储什么类型的数据，而且不难想象，有时候用户会希望他们的特性长于500个字符。因此，至少现在，我们会倾向于使用较大特性值。

这样，我们可以创建文件系统资源模型的第一个版本。模型的基本结构如下所示，该代码中尚没有实现其行为的方法：

filesystem/first-persistent.py

```
class PersistentResource(db.Model):
    content = db.BlobProperty(default = "")
❶   attributes = db.ListProperty(item_type=db.Key)

    @staticmethod
❷   def MakeResource(creator):
        resource = PersistentResource()
        resource.content = ""
        attribute = ResourceAttribute(name="creator", value=creator)
❸       attribute.put()
❹       resource.attributes.append(attribute.key())
        resource.put()
```

❶ 资源属性的列表特性非常简单。因为我们定义了一个数据仓库知道如何持久性保存的属性类型，所以可以为对象创建一个关键字列表。因此，只需定义列表特性作为数据仓库关键字列表。就这么简单。

❷ 为数据仓库的持久性类型提供标准的Python的`__init__`方法是一个非常糟糕的主意：数据仓库的实现提供了一个默认的初始化过程，正是因为用户无法改变数据仓库内部行为，所以才保证了它的正确性。因此，我们使用静态方法构建内容。在静态`MakeResource`方法中，创建一个资源的空实例，然后填充几个字段。创建资源模型的实例时，将初始化一个属性。我们并不一定要做这项工作，客户端不会直接调用此代码。它们只能通过稍后构建的HTTP接口调用。但是为了说明此代码是如何工作的，我们在这里初始化`creator`特性。

❸ 按照正规的方式创建属性，使用数据仓库提供的默认的构造函数。然后，告诉数据仓库存储该属性对象。这样做是有原因的。首先，属性不包含在资源对象中，因此存储资源时，不会存储属性对象，只会存储一个对属性对象的关键字引用。因此，属性需要被存储。我们需要在存储资源前存储属性，这是因为要将属性放到资源的属性列表中，需要使用它的关键字，而关键字直到对象存储时才会被初始化。因而，创建出属性，并先将其放到数据仓库中。

❹ 有了存储的属性，现在可以访问其关键字了。要设置资源的属性，只要在属性列表中添加属性对象的关键字就可以。然后，我们存储该资源，所有的事情就都做好了。

现在，我们有了恰当的模型，需要看看如何实现它们的行为。基本的内容的get和put简单易懂：

filesystem/first-persistent.py

```
def GetContent(self):
    return self.content

def PutContent(self, content):
    self.content = content
    self.put()
```

根本不用任何解释，应该很清楚！现在，可以把重点转移到属性上。让我们先看看如何取回属性的值：

filesystem/first-persistent.py

```
    def GetAttribute(self, name):
❶       for attr_key in self.attributes:
❷           attr = ResourceAttribute.get(attr_key)
            if attr.name == name:
                return attr.value
        return None
```

❶ 资源的attributes字段是一个标准的Python列表。该列表中的元素是属性值的关键字。要找到一个具有特定名称的属性，我们需要遍历所有的属性。可以只用一个标准的Python循环实现特定名称的属性查找。

❷ 现在，在这个循环中，我们要查看实际的属性。因此，需要做的第一件事情是取回属性。当用户有关键字时，可以通过调用Classname.get(key)取回属性。从这里，读者就会明白，为何即使用了关键字，也不能实现一个可以包含不同类型元素的列表。当用户通过关键字取回存储的对象时，需要知道它的类型，以调用该关键字。因此在这里，由于我们知道关键字都是针对ResourceAttribute对象的，所以可以通过调用ResourceAttribute.get(attr_key)来取回对象。

开发者可能会担心进行一系列get操作的性能，因为每个属性需要一个get操作。说真的，其性能还不错。首先，每个资源属性个数都很少，在使用这种结构的实际文件系统中，资源的最大属性数量通常是20个左右，大多数都比这少。这意味着取回的数量非常少。此外，通过关键字取回真的很快，比通过查询语句取回要快出很多。关键字取回和查询取回的具体关系取决于查询的复杂性，但通常通过关键字所做的很多次取回操作要比单个查询语句的成本要低。

SetAttribute的实现非常相似：

filesystem/first-persistent.py

```
    def SetAttribute(self, name, value):
❶       for attr_key in self.attributes:
           attr = ResourceAttributeModel.get(attr_key)
❷          if attr.name == name:
               attr.value = value
               attr.put()
               return
❸       newAttr = ResourceAttribute(name=name, value=value)
        newAttr.put()
        self.attributes.append(newAttr.key())
        self.put()
```


❶ 就像在`getAttribute`方法中一样，需要搜索属性列表，找到要改变的那个属性。因此，用一个完全类似于`get`中所做的循环：遍历了属性关键字，并对于每一个关键字，取回其属性对象。然后再检查它的关键字。

❷ 如果找到一个属性，其关键字与要设置的匹配，只要更新该属性对象并重新存储即可。由于在原位更新值，所以并不需要担心重新保存关键字，属性关键字已经存储在资源对象中，而且并没有改变它。一旦完成更新，就大功告成了，因此，可以返回。

❸ 如果该特性不存在于资源中，那么将退出循环而不必返回。在这种情况下，需要创建该属性，并把它保存到数据仓库中。然后就可以把该属性的关键字放到资源的属性列表中，并且保存更新的资源。这种情况下需要保存资源，因为增加了新的属性，从而修改了该资源。

综上所述，这是文件系统的基本核心功能，以数据仓库可以使用的方式实现。还需要做一些其他事情，比如说希望能够查询一个给定的资源是否是一个目录。按照数据模型，目录是一个包含子属性的资源。这种检查很容易实现：

filesystem/first-persistent.p

```
def IsDir(self):
    return self.GetAttribute("children") is not None
```

还需要处理如给目录添加新文件这样的事情。如何才能实现这个功能呢？我们想当然地会从资源中获得`children`属性，然后在其中添加一个文件：

filesystem/first-persistent.py

```
def NaiveAddChildToDirectory(self, name, resource):
    children = self.getAttribute("children")
    if children is not None:
        children.value.append(resource) # WRONG!
```

问题是这并不能正常工作。如前所述，有些值需要特定的类型。属性的值的类型是文本，因而不能仅把资源对象放在那儿。那么，我们如何表示子列表呢？如果`children`真的只是另一个属性，那么它不应被视为与其他属性有任何区别。但是，这确实会很棘手。那么，我们能做些什么呢？

这个问题并不容易解决，因为这涉及到设计一个良好的数据仓库模型的一个关键因素：开发者把不同对象之间的分界线划在哪里？以及，开发者如何描述对象之间的关系？在数据仓库中，开发者通过引用实现此功能。

13.2.1　数据仓库关键字和引用

当涉及在数据仓库中处理`children`属性的值时，有两个切实的选择：可以将目录结构编码为字符串，然后再将该字符串放到属性中，或者可以将`children`属性看作一个特例。稍后再来对比这两种方法具体的优劣之处，现在将只选择简单的方法，将`children`作为一个特例。目录结构是文件系统中的关键细节，因此，虽然它似乎有点麻烦，但其重要性足以值得去特殊对待。

但无论采用哪种方式，都会违背另一个重要因素。目录中的文件是一个独立的对象。如果按照传统的对象模型进行思考，文件并不是包含它的目录的一部分。它是一个独立的对象，被它的父目录所引用。在类似数据仓库的系统中，这种区分尤为重要。

假设认为文件是包括它们的目录的一部分，那么，在从数据仓库取回文件系统的根目录时，就需要从数据仓库载入整个文件系统。根目录最终包含系统的每个文件和目录，如果根目录只包含其中一部分，这部分也将使用根目录来进行存储和取回。另外，我们甚至不能取回一个单独的文件，因为它只作为根目录里的一个属性存在。

这绝对不是我们想要的。

文件系统应该是一个由各独立对象组成的集合。按照经典的内存中的对象模型，我们要的是一个指针。我们不希望目录包含其子节点，而希望它指向它的子节点。

数据仓库中也有类似于C++中指针的东西，它被称为引用（reference）。引用不同于C++的引用，C++的引用其实只是指针的另一个名称。对象存储（object-store）的引用是一个标识符，该标识符唯一标识了数据仓库中的某个对象。引用就像是数据仓库中指向目标对象的指针，它不是指向它的位置，而是提供了一个标识符，通过该标识符数据仓库能够取回这一特定的对象。

正如之前提到的，数据仓库中存储的每一个对象都包含一个唯一的标识符，称为关键字（key）。对另一个对象的引用是一个特性，该特性的值是另一个对象的关键字。到目前为止，我们在以一种非常原始的方式使用关键字。然而，App Engine为我们提供了另一种便捷的使用关键字的机制。数据仓库中的引用对象是对关键字的封装，使得该对象好像是被关键字本身所标识一样。当开发者尝试访问一个引用对象的字段或方法时，它自己会自动进行取回，然后将调用转发给取回的对象。就我们的文件系统而言，这意味着可以返回一个特定的目录中的所有子对象的列表，而不用真正取回所有子对象。

来看一下如何实现上述功能。创建一个新版本的资源模型类型，它有一个子节点专有的特性。就像为属性模型所做的一样，子节点的特性是对象列表，其中每个元素都是一个“名称/值”的二元组：

filesystem/persistent_filesystem.py

```
❶ class DirectoryEntry(db.Model):
```

```
    name = db.StringProperty()
❷   resource = db.ReferenceProperty(PersistentResourceModel)
```

❶ 需要定义一个目录项的模型类，也就是一个典型的db.Model的子类。

❷ 该句声明了一个特性，即对另一个资源的引用。为了生成引用特性，用db.Reference值创建一个字段，并为构造函数提供一个db.Model子类的参数。当把该类的value字段设定为特定的资源时，会保存该资源的关键字。

现在，可以编写资源定义了。我们要做的是使children成为一个特例——我们固定将children属性连接在该资源上，这是一个DirectoryEntry关键字的列表。

filesystem/persistent_filesystem.py

```
class PersistentResource(db.Model):
    content = db.BlobProperty(default = "")
    attributes = db.ListProperty(db.Key)
❶   children = db.ListProperty(db.Key)
```

❶ 在新的资源模型中，子节点包含了一个特性，是DirectoryEntry对象的关键字的列表。我们不使用对该条目的引用，不能在列表特性中这样做。此外，从概念上讲，我们希望在资源对象中包含这些条目，这就意味着，只能从资源类的方法内部存储或取回这些条目。引用的要点是，当真正的取回发生时，引用使取回过程透明。然后，开发者可以传递引用，好像它是真实的对象，而无需担心真正需要它的值的时候如何取回它。但是，在这里，只有一个地方会取回这些引用值，所以引用的开销就无需考虑了。

现在我们已经有了一个有趣的设置。资源类型中包含一个目录项的列表，该列表按照关键字访问，且目录项是引用。我们如何将此设置应用于实现目录呢？首先，来看看需要如何修改getAttribute和setAttribute方法，以便使用目录项。然后再来看看使用这些文件系统对象的代码，了解我们实际上是如何使用引用的。

filesystem/persistent_filesystem.py

```
def GetAttribute(self, name):
❶   if name == "children":
        return [ DirectoryEntry.get(key) for key in self.children ]
    else:
        for attr_key in self.attributes:
            attr = ResourceAttribute.get(attr_key)
            if attr.name == name:
                return attr.value
        return None

def SetAttribute(self, name, value):
    if name == "children":
❷       self.children = [ de.key() for de in value ]
        self.put()
    else:
        for attr_key in self.attributes:
            attr = ResourceAttributeModel.get(attr_key)
```

```
                if attr.name == name:
                    attr.value = value
                    attr.put()
                    return
                newAttr = ResourceAttribute(name=name, value=value)
                newAttr.put()
                self.attributes.append(newAttr.key())
                self.put()
    def IsDir(self):
        return (self.children is not [])
```

❶ 在GetAttribute方法中，我们先检查一下属性名称是否为children。虽然这种检查令人生厌，但由于处理的内容的局限性，所以它也是不可避免的。children属性是一种特例，因此需要显式地检查。如果它就是要取回的内容，那么我们使用特定代码取回资源的子节点，否则，就直接跳到与之前使用的相同的代码处。如果正在取回children属性，那么就使用列表遍历：遍历DirectoryEntry关键字列表，取回实际的DirectoryEntry对象，然后返回该列表。因此，对于我们的文件系统的客户端而言，当它们调用file.GetAttribute("children")时，得到的结果就是一个目录项列表。但是这里隐藏了一点小技巧：正如后面将会看到的，对于这段代码的客户端，目录项的行为就好像它们确实包含子节点资源一样，但实际上，它们只包含了引用。

❷ setAttribute方法与GetAttribute方法大体相反。我们以完全相同的方式开始，首先检查设置了什么属性。如果是children，那么我们希望其值是一个DirectoryEntry的列表，所以使用列表遍历，实现与get完全相反的工作：用关键字替换目录项，获得关键字列表，然后将其存储到资源特定的children属性中。因为该操作修改了资源，所以我们需要使用put调用保存。

当引用的属性设置好之后，我们如何使用它呢？数据仓库完成了所有的工作。如果只是要获取数据仓库中引用实例的特性值，数据仓库会自动取回相应内容，其过程对开发者完全透明。因此，举例来讲，我们可以使用下面的函数遍历目录中的子节点，打印出每个子节点是否是目录：

filesystem/persistent_filesystem.py

```
def RenderChildren(dir):
    children = dir.GetAttribute(children)
    for c in children:
        # c是DirectoryEntry
❶       if c.resource.IsDir():
            print("Child %s is a directory" % c.name)
        else:
    print("Child %s is not a directory" % c.name)
```

在该函数中，遍历目录的children特性中的DirectoryEntry值。对于每一个值，通过对DirectoryEntry的resource字段调用IsDir方法，访问其引用的资源。这个字段不是一种资源，它是一个引用。但每当试图访问引用的任何字段或方法时——正如这里，当调用它的IsDir方法时——数据仓库会自动取回被引用的对象。

到目前为止，实现似乎非常简单——几乎很简单的持久性也会变得极其复杂，难以获得正确结果，但是到目前为止，我们所看到的内容完成了这个令人钦佩的工作，隐藏了其中的复杂性。然而，潜在的复杂性依然存在。

这种潜在的复杂性会从几个方面影响开发者。最明显的一点就是，数据仓库在我们使用引用时，自动取回了引用，这虽然不错，但实际上它仍然是一个对数据仓库的往返调用。从持久性存储系统中取回内容并不是免费的。虽然这个过程并不是非常昂贵，远不如做一个查询昂贵，但它不是免费的，而且这样做的成本会逐渐积累，最后变得相当显著。把取回操作对我们隐藏，这样做固然方便，但也很容易让代码突然变得很慢。

例如，在上边打印目录的程序里，按照代码中的方法进行目录迭代时会发生什么事情呢？该程序取回了目录中的每个子节点的资源对象。这样做，意味着取回了目录中的每一个成员的每一个子节点的每一个DirectoryEntry的关键字。如果使用这些代码，最终会取回其他资源对象的引用。编写一个遍历程序遍历目录层次结构确实很容易，但是会做大量的取回操作，每个取回操作都独立作为引用访问的一部分。在这样的情况下，我们用一个单一的查询语句一次取回所有对象会更好。如果开发者不习惯云风格的分布式编程，那么，这种对引用的随意使用，最终会导致潜在效率非常低的代码，只是效果不很明显而已。

通过下面这个例子，开发者就能理解其潜在影响。在早期的职业生涯中，我曾做过一些C++持久化系统的相关工作，跟自动引用取回操作很像。有一次，我为客户做一些示例代码，发现它的性能完全不合理。一个循环迭代的时间花费非常巨大。它并不与列表的大小大致成正比，而是与列表大小的平方成正比。这个问题恰好证明了我在上文中所描述的问题：每次有人访问列表的成员，它会自动取回其子节点列表，并为子节点建立代理（引用的形式）。因此，访问列表元素实际上最终创建了一个代理列表。看起来没有影响的一行代码，由于系统自作聪明的幕后操作，实际上代价非常昂贵。

在数据仓库中，开发者需要知道到底发生了什么，以理解其各项花费。当开发者访问一个引用参数时，真正做的是对数据仓库进行一个调用：db.get(key)。这会引发一个非常昂贵的取回操作。

有时候这是一个很现实的问题，但是，开发者可以通过显式地使用关键字来避免此问题。使用关键字时，正如我们使用DirectoryEntry对象的实现一样，可以显式地进行取回操作。这样，因为取回操作不再对开发者隐藏，在代码中，进行了多少取回操作变得很清晰。在这种编程中有一个微妙的平衡，即在使用简单的内容（如引用）和显式的内容（如关键字）中进行选择。在一般情况下，如果开发者有很多对象，我建议使用显式的关键字。虽然这样要付出更多努力，但是能让你更清楚地知道程序的行为。

构建文件系统的下一个主要问题是：应该如何处理内容？在初始模型中，我们在一个文本特性中存储内容。这很好，但不是非常好。文本对象可以相当大，但它们仍然基本上只是字符串。这对文件系统来说并不完全正确：一个文件系统中的文件的内容可以任意大，而且其内容可以是任何字节序列。

但还有另外一个问题。每次取回文件对象时，取回的是整个文件对象。如果其内容是20兆字

节，并且我们想要做的工作仅仅是检查该文件是否是一个目录呢？就如同之前写的显示函数里所做的事情一样，这种实现的效率低得难以想象。请记住：在云中，开发者要为使用的东西付费。开发者加载其不使用的文件的内容所花费的时间并不是免费的。我们真正要做的是为内容创建另一个对象类型，这样就可以使用引用了。

最终的对象模型（对本节而言）如下：

filesystem/filesystemblobmodel.py

```
❶ class ContentModel(db.Model):
      data = db.BlobProperty()

  class DirectoryEntry(db.Model):
      name = db.StringProperty()
      resource = db.ReferenceProperty(PersistentResourceModel)

  class Resource(db.Model):
❷     content = db.Reference(ContentModel)
      attributes = db.ListProperty(db.Key)
      children = Db.ListProperty(DirectoryEntry)
```

这与之前的内容的确很接近。主要的区别如下所述。

❶ 为文件内容创建了一个新的类。它唯一的属性是`Blob`。除了不是解析为字符串序列的一系列字节，它很像文本，它只是一个完全不解析的字节序列。对于文件，这正是我们想要的：读取文件内容的程序可以决定如何解释该`Blob`。

❷ 在`Resource`类中，我们使用了对内容对象的引用。有了它，取回资源对象时，我们只要得到其内容的引用即可。只有试图访问内容时，内容才会被取回。

13.2.2　实现文件系统的其余部分

我们已经处理了文件系统中大部分数据仓库，但仍然需要用一些粘合代码把各部分组装起来。到目前为止，我们所做的是实现文件和目录，因此需要将它们组合成一个完整的文件系统。

最终，文件系统是文件和目录的集合，其中有一个特殊的目录，称为根目录（root）。所有对文件系统内文件的引用都会按照相对于根目录的方式来计算，因此文件系统需要做的另一件工作已经隐含在上句话中，即文件系统需要能解析它所引用的特定文件的全路径，如果它不能解析，就会产生错误。

那么，怎样实现一个最低限度的文件系统呢？参看如下代码。其中，唯一复杂的部分是`GetResourceFromChildByList`的实现，该函数递归遍历目录层次结构，取回一个特定文件。如果对下面的代码有疑问，请暂时不用担心，这段代码对理解接下来的内容并不是很重要。它所做的全部工作就是从像`/a/b/c.txt`这样的路径开始，首先在根目录查找名为a的资源，然后在资源a中查找名为b的子节点资源，然后再在b资源中查找名为`c.txt`的子节点资源，最后返回`c.txt`，即找到正确的资源。

filesystem/filesystem_servlet.py

```
class Filesystem(db.Model):
    root = db.Reference(Resource)

    def GetRoot(self):
        return self.root

    def GetResource(self, path):
        path_elements = path.split("/")
        return self.getResourceFromChildByList(self.root, path_elements)

    def GetResourceFromChildByList(self, resource, path_elements):
        if path_elements is []:
            return resource
        else:
            for direntry in resource.children:
                if direntry.name == path_elements[0]:
                    return getResourceFromChildByList(direntry.resource,
                                                      path_elements[1:])
            return None
```


13.2.3 用GET实现文件获取

解决了基本的文件系统模型之后，接下来要做的就是实现该文件系统。文件系统的数据仓库的各部分都很容易，至少一开始是这样。要实现一个基本的的文件系统，几乎就是一系列的有着完整对象的GET和PUT。实现URL映射有一些工作量，但这与我们之前看到的并没有什么差别。不过，我们将会看到，基于现有的这些东西就可以做一些更好玩的事情，如基于属性值在文件系统中进行查找操作。

先从基础开始。用户能用文件系统做些什么呢？

(1) 创建资源。这涉及到创建资源对象，设置其属性，存储其内容，并修改其父节点资源，即把这个新的资源提供给一个目录项。

(2) 获取资源的内容。

(3) 设置和获取资源的属性。

(4) 更新资源的内容。

我们必须将这些动作映射到基本的HTTP操作上：GET、PUT和POST。获取资源的内容或者它的属性显然是一个GET操作，创建一个新的资源或者更新其内容为PUT操作。

事实上，现在我们不需要POST，我们将使用GET来实现所有的获取，使用PUT实现所有的存储。

现在，需要思考的是如何组织URL。如果忽略属性，那么这很容易。URL中有简单的路径，而且标准的URL语法也正是为此设计的。但对于属性则有点棘手。可以使用CGI参数来修改HTTP调用，或者可以创建一种将属性编码到URL的方式。经典的REST风格提倡后者的做法。属性是一种可以被服务访问的数据，每一块可被唯一识别的数据都应该有它自己的URL。

属性是资源的元数据位，所以它们的URL应该基于其相关文件的URL。为了防止混淆文件的属性和目录的成员，我们将在属性前加前缀“~”，这样我们就不会认为是资源名称的路径字符了。即，给定一个URL为R的资源，它的属性a1将通过R/~a1的URL进行访问。

可以用GET处理程序来生成文件系统服务的第一个版本：

filesystem/filesystem_servlet.py

```
class FilesystemResourceHandler(webapp.RequestHandler):
    def GetFilesystem(self):
        query = Filesystem.gql("")
        return query.get();

    def get(self):
❶       filesystem = self.GetFilesystem()
❷       root = filesystem.root
❸       url = self.request.path
        urlElements = url.split("/")
        # 然后检查它究竟是请求资源内容，还是请求属性。
        #如果最后一个名称元素的首字符是"~"，那它就是一个属性
        resourcePath = None
        attr = None
❹       if urlElements[-1].startswith("~"):
            attr = urlElements[-1]
            resourcePath = urlElements[:-1]
        else:
            resourcePath = urlElements
❺       resource = filesystem.getResourceFromChildByList(root, resourcePath)
❻       if resource is None:
            self.response.error(404)
            return
❼       if attr is not None:
            result = resource.getAttribute(name)
        if result is None:
            self.response.error(404)
        self.response.out.write(str(result))
        return
    else:
❽      self.response.out.write(resource.content.data)
```

❶ 首先，对于云应用程序而言，通常需要调用取回功能，获取文件系统的根对象。我们知道该如何编写该功能，这只是一个简单的数据仓库取回。

❷ 通过文件系统，可以得到根资源，也就是文件系统的根目录。

❸ 然后来看请求，从其中获取想要取回的资源的路径。

❹ 解析该路径，看看它是否是对一个属性或者资源内容的请求。

❺ 使用之前在上一章中写的方法，取回资源对象。

❻ 如果获取资源的尝试失败，则返回空值。因此产生一个404响应（即，HTTP对“未找到资源”的响应）。

❼ 如果URL中包含属性名称，取回该属性，并在结果消息的内容中返回属性。如果不指定

结果代码，它会默认为成功的代码，而且content-type会默认为UTF-8文本。由于这些都是我们所希望的，所以，只需将属性值写入响应数据流即可。

❽ 否则，我们使用数据仓库的引用自动取回功能，获得资源的内容，并将其写入数据流。同样，默认值能很好地工作：当我们向响应输出流写一个blob时，如果该blob只是文本字符，它会默认为UTF-8；如果它有任何非文本字节，它会默认为字节流。

13.2.4 用PUT实现文件存储

在实现文件存储中，值得关注的事情是，当开发者PUT文件时，可能还需要更新其父文件，从而为新资源创建一个新的目录项。

PUT的代码与GET非常相似。我们从解析路径开始。如果PUT是内容更新，则需要获取资源，如果资源尚不存在，就获取其父资源。如果是一个属性更新，资源必须存在，所以我们需要先获取资源本身，然后，做相应的更新。

filesystem/filesystem_servlet.py

```
    def put(self):
        filesystem = GetFilesystem()
        root = filesystem.root
        url = self.request.path
        urlElements = url.split("/")
        resourcePath = None
        attr = None
        if urlElements[-1].startswith("~"):
            attr = urlElements[-1]
            resourcePath = urlElements[:-1]
        else:
            resourcePath = urlElements
         resource = filesystem.getResourceFromChildByList(root, resourcePath)
❶        if resource is None:
             parent = filesystem.getResourceFromChildBylist(root,resourcePath[0:-1])
             name = resourcePath[-1]
             if parent is None:
                 self.response.set_status(404, "Parent dir of new resource not found")
             else:
                 resource = Resource(content = self.request.body, attributes=[],
❷                                    children=[])
                 resource.put()
❸                dirEntry = DirectoryEntry(name=name,resource=resource)
                 parent.children.append(dirEntry)
                 parent.put()
                 self.response.set_status(100, "Resource created")
                 return
          else:
❹            resource.content=self.request.body
             resource.put()
             self.response.set_status(100, "Resource updated")
```

❶ 在尝试取回资源的地方之前，一切都和GET的过程很像。在GET中，如果不能找到所请求的资源，就会返回一个错误，而在PUT中，则尝试创建该资源。为了创建它，需要获取父资源。如果找不到父资源，就返回一个错误。但如果能找到父资源，就继续创建新的资源作为它的子资源。

❷ 通过实例化模型类，可以创建一个新的资源。该模型的每个特性是构造函数的一个命名参数。资源被创建后，我们使用put()将它保存到数据仓库。

❸ 新的资源创建并存储后，我们需要在它的父对象中为其创建一个目录项，把新资源加到它的父对象资源中，并使用put保存更新后的父资源。

❹ 如果资源已经存在，则只需更新其内容，然后对该资源利用put方法。

13.3 特性类型引用

Google App Engine提供了相当多数据仓库模型的特性的支持。在本节中，我们将浏览一系列特性。这有点枯燥，但是，有这么一个完整的列表方便今后查询，是很方便的。数据仓库支持的特性，大致分为两种类别。一种是原始类型，对于在数据仓库中存储东西非常重要的基本类型；另一种是效用类型，这是更加专用的东西（例如邮箱地址），由于这些类型经常出现，所以提供了标准化的格式和验证方法。

13.3.1 原始特性类型

- `BlobProperty`
  blob是一个“二进制的大型对象”，换句话说，一个任意大小的字节块。开发者不能在查询语句中使用任何有关blob的东西，它们没有可比性，也无法排序。blob特性的值是`db.Blob`的实例，这是一个`str`的子类。因此，开发者可以在其代码中使用blob，就好像它们是字节串，因为它们确实就是字节串！
- `BooleanProperty`
  `BooleanProperty`是一个单一的布尔值，True或者False。
- `ByteStringProperty`
  基本上是与blob同样的东西，但有索引，并像字符串一样长度有限。
- `DateProperty` / `DateTimeProperty` / `TimeProperty`
  时间戳对象。在Python中，作为`datetime.datetime`的一个实例实现。它实际上有三个版本，大致是相同的，因为都是`DateTime`对象。使用`DateProperty`，时间字段保留为空；使用`TimeProperty`，日期字段为空。它们在查询结果中按时间顺序排序。
- `FloatProperty`
  一个浮点数。
- `IntegerProperty`
  一个64位整型值。

- Key

  该特性用来存储唯一区别于其他数据仓库对象的关键字。关键字应该是完全不透明的，它们不可排序，除了相等之外无法进行其他任何比较。如果开发者确实需要，那么有分解关键字提取信息的方法，但是，大多数时候，如果开发者需要进行关键字信息提取，只能说明这是持久化设计具有严重问题的征兆。

- ListProperty

  数据仓库支持的一系列数据类型。它将列表元素的类型作为参数。不同于通常的Python代码，开发者不能有混合类型的列表，数据仓库列表特性只能存储个单一类型的值。此外，还有StringListProperty，这只是ListProperty(item_type = basestring)的缩写。

- ReferenceProperty

  对于某特定类型对象的引用。被引用对象的类型是构造函数的一个参数。当开发者取消一个ReferenceProperty时，数据仓库会自动地、透明地进行get操作以取回对象。

- SelfReferenceProperty

  一个对象包含了对同类型对象引用的引用特性的便捷缩写。如果资源中有一个字段是其他资源，我们可以使用SelfReferenceProperty来定义它，而不是ReferenceProperty(Resource)。

- StringProperty

  一个字符串值。

- TextProperty

  TextProperty是字符串和blob之间的交集。这是一个字符串类型，其值解析为Unicode字符串。但是，像blob一样，它可以是任意大小，并且它的值不能在查询语句中使用。

13.3.2 复杂特性类型

- CategoryProperty

  一个字符串特性。该特性在构建像Atom订阅这样的内容中使用，它代表一个atom目录。

- EmailProperty

  字符串的一个子类，它代表电子邮件地址。与大多数专用类型不同，该类型没有提供邮件地址的语法验证。它唯一有用的是作为给阅读开发者代码的人们的一个提示，其特性值会被解析为电子邮件地址。

- GeoPtProperty

  一个地理位置。该值是一个XML元素，包含标准的GEORSS表示形式的地址，其定义参见http://georss.org。

- IMProperty

  即时消息句柄的表示。该特性可以被Google App Engine服务用于即时消息服务的通讯。它从标识即时消息协议和用户名的URL构造而来。例如，我的Google Talk的即时消息将

是`db.IM("http://talk.google.com", "markcc@gmail.com")`。

- `LinkProperty`

  `LinkProperty`是一个字符串，其中包含一个有效的URL。这是设计用于Atom订阅的，但是，它对于任何需要存储链接的应用程序普遍有用。它提供了对URL的充分验证。

- `PhoneNumberProperty`

  一个包含电话号码的经过验证的字符串。该特性针对各种国际标准的电话号码格式进行了验证。

- `PostalAddressProperty`

  一个字符串值，其中包含邮政地址。

- `RatingProperty`

  一个标识用户等级或排名的值。它是从0到100的一个整数。

13.4　特性类型结束语

在这一章中，我们深入了解了数据仓库。学习了可以在数据仓库中使用的特性类型，看到了在存储的对象中容器和引用之间的差异，学习了关键字和引用，这是在数据仓库中管理存储对象之间的关系的主要机制。利用这些内容实现了数据仓库中的文件系统服务的基本结构。

关于数据仓库，还有很多要学习的内容。为了生成一个高效的、可扩展的数据仓库模式，我们需要知道如何定义索引，以及如何选择并创建合适的索引，以支持希望执行的查询语句。当然，还有很多可以使用模型完成的工作。到目前为止，已经看到的模型有一组固定的特性。数据仓库确实允许我们定义模型，并且可以根据需要添加新的特性，但是这样做有其自身的一些限制。

下一章将介绍一些更高级的数据仓库功能。我们将重点关注查询问题，如索引、关键字、游标和策略，以及查询语句的工作原理。另外，还将介绍如何能够创建更加灵活、可扩展的模型。同时，我们会通过增加查询特定特性的资源的方式，在文件系统实现上增加更多功能。

第 14 章

高级数据仓库：特性类型

14

在上一章中，我们了解了可以在Google App Engine数据仓库中用作特性的值的类型。虽然主要用的是Python来处理，但该特性类型集对Java和Python都是相同的——也就是说在Java和Python中都能够作为可存储对象的字段的值的类型。

本章将基于此来介绍数据仓库中查询实际的工作原理。在后台，使查询工作快速、可扩展的过程是使用索引完成的。App Engine的SDK可以根据用户的代码为其数据自动生成索引，但有时用户自己定义索引可以带来更好的性能，并使用更少的索引。同样，像上一章一样，我们大部分工作用Python完成，但是在Python和Java中，查询和索引工作完全相同，也就是说，无论用户使用哪种语言，都需要为同种类型的查询按照完全相同的方式创建自定义索引。

在这一章中，我们将了解数据仓库如何生成索引，以及如何通过在`app.yaml`中声明索引的方式定义自己的索引。我们将用该方法定义文件属性的索引，从而能够搜索文件系统，另外，将最终形成部署文件系统应用程序所需的完整的`app.yaml`文件。

14.1 数据仓库中的索引和查询

在深入App Engine的索引工作原理的细节之前，值得先花一些时间学习一些幕后的技术。基于这些索引技术的数据仓库索引和数据仓库查询，与大多数人习惯使用的类似方法完全不同。数据仓库的索引有很多限制，除非开发者知道究竟是什么原因造成的，否则这些限制看起来简直很武断。

14.1.1 揭开数据仓库的面纱

最终，我们存放到数据仓库的所有数据实际上是使用称为Bigtable的Google存储系统存储的。虽然开发者并不直接对Bigtable编程来使用数据仓库，但是，Bigtable定义了数据仓库的存储、查询，以及性能的基本特性。

如果不深入太多细节的话，那么，Bigtable所提供的用于存储数据的结构与开发者听到“表”的时候可能想到的内容并不完全相同。Bigtable的存储模式很有技巧。在某些方面，它提供了表，但是，这些表并不像关系数据库的表。人们使用Bigtable和数据仓库工作时最常见的错误就是像在关系数据库中一样设置数据。在关系数据库中，开发者做很多的工作以提高性能和可查询性，

例如数据标准化。但是，如果开发者在数据仓库中进行数据标准化，那么，着实会影响程序的性能，并且查询语句的编写会变得更加困难！

数据标准化是关系数据库的编程人员喜欢使用的一种技巧，因为在关系系统中数据标准化能够带来绝对巨大的益处。其思想是，开发者尝试把所有内容推倒，使每个单独的表尽可能是扁平的：复杂数据结构以表之间的关系形式存储。例如，如果开发者有一个对象（如我们的`Resource`资源对象），该对象包含了一组值的集合（如我们的`Attribute`属性值），那么，开发者会将这些值通过一个中间表分隔。也就是说，开发者需要从资源对象的存储模型中彻底删除属性，并且要为每个属性添加一个唯一的标识符。然后，开发者会在单独的表中存储`Resource`和`Attribute`值，并添加第三个表，称为关系表，关系表中包含了资源标识符和属性标识符的组合列表。因此，要检索资源的属性，开发者可以进行查询，类似于`select attr from attributes, rel where attr.id = rel.attr_id and rel.resid = ?`。

而在数据仓库中，开发者*真*的不希望这样做。

与关系数据库的表比起来，Bigtable更像是两级的散列表或者查找表。

大多数人认为的表是一个矩形网格，行和列有标签。每行具有完全相同的列。这就是关系数据库中使用的表的模型。

Bigtable有行和列，但它们与关系数据库的行和列不同。在Bigtable中，所有数据都以行存储。每行只有一个关键字。快速访问行的唯一途径是知道其关键字。反过来，每一行由一组列组成，但列完全是任意的。不同的行可以有完全不同的列。而且，每一行可以有成千上万个不同的列！

这就回到了前边我所指的Bigtable是两级散列表的意思。给定表中的一行的关键字，开发者可以非常快地取回该行。也就是说，在顶层，Bigtable的功能类似于散列表或者字典，让开发者快速查找与特定的关键字关联的行。一旦开发者获得该行，就好像获得了一个列的散列表，即只要开发者具有行和列的标识符，就可以非常迅速地从散列表中取回某个特定的列。

除了查找工作之外，事务都是以行为中心的——每个事务都是保护单一行上的操作。每次当开发者进行涉及多个Bigtable行的数据仓库的更新操作时，就会大大增加事务的复杂性。

当然，开发者并不确切知道其模型是如何映射到一个Bigtable上的。事实上，随着时间的推移，开发者的数据映射到Bigtable的方式可能会发生改变。但是，映射的主体架构是非常直观的。

在数据仓库中，每一个模型对象是Bigtable的一行。即使开发者采用了列表字段——如我们的资源对象的属性字段——在对象中保留该字段，意味着将在相同的Bigtable行中保留这些值，这会有显著的性能优势。因此，对于数据仓库中的性能，开发者实际上要做的工作与标准化相反，即，并不是尽可能地将对象扁平化，而是要把它们捆绑起来！

类似地，查询的工作方式也与Bigtable相关。Bigtable支持基于比较对一系列值进行搜索的方法。但由于Bigtable的实现方式，这种方法受到了限制。所有的查询限制都来自于Bigtable的基本特性。因此，举例来说，在任何查询中，开发者只能在一个字段上进行关系比较。这是因为关系比较只有通过索引才能高效实现，而开发者在Bigtable的查询中只能使用一个索引。

14.1.2 自动生成的索引

正如我前面所说的，App Engine的SDK可以自动生成数据仓库的索引。基本上，当开发者首次执行一个dev_appserver.py提供的使用本地App Engine执行环境的特定查询时，App Engine就会检查开发者的存储模型，看看是否有适合该查询的索引。如果有，那么它就使用该索引；如果没有，它会创建一个。

对于大多数应用程序而言，这种处理方法很好。开发者不必担心索引，数据仓库会为开发者处理好它。开发者只要根据其应用程序的自然特性来定义模型（切记，当然，开发者是为数据仓库定义模型，而不是为关系数据库！），数据仓库的基础设施会尽力使开发者的查询快速运行。

但往往自动的东西并不完美。App Engine和数据仓库只是尝试着为某些它明确知道可以正确索引的情况自动生成索引。如果开发者想在此范围外工作，就需要自定义索引。

那么什么情况下数据仓库知道如何做正确呢？有以下4种常见情况。

- **等式过滤器**
  数据仓库为所有字段内置了等式索引。如果开发者需要的是基于等式比较的查询——也就是说，任何整个过滤语句为`WHERE x = y`（或由AND连接的一系列等式比较）的查询语句，那么，开发者不需要担心生成索引的问题，数据仓库总会有其索引。
- **无过滤器，有序的**
  如果开发者正在做一个查询，其中没有比较，而且只有一组按照单一顺序进行排序的值（例如，一个返回按日期排序的所有聊天消息的查询语句），那么，App Engine可以自动生成索引。
- **对象等式，特性关系式**
  对于最上面一级的持久性对象之间的所有比较都是等式测试，或者特性之间的所有比较都是相同特性的关系比较的查询语句，App Engine可以自动生成过滤器。
- **关系式过滤器**
  如果开发者的查询比较中唯一的过滤器是关系比较（即小于或大于），并且所有的比较都是同一特性的，那么，App Engine可以自动生成索引。

虽然上述内容看似有限，但它们确实覆盖了开发者常会遇到的大多数情况。例如，在我们的聊天应用程序中，曾使用了查询语句，比较了一个消息的chat特性与一个特定值，以便在聊天室中显示该消息。那是一个等式比较——因此，根据上述第一种情况，App Engine会自动索引。

事实上，到目前为止，我们写的每一个查询语句都属于缺省索引。缺省索引已经足够用了，几乎我们想写的每个查询都可用缺省索引表达。所以找出一个使用自定义索引的例子确实也比较难！

14.1.3 创建自定义索引

正如之前所说的，在大多数情况下，开发者可以绕开自动生成索引的种种限制。此时，就应该使用自动索引。这些限制都是有原因的。当扩展到大量数据时，很多相关因素就会带来潜在性

能问题。例如，我们将要用到的查询要求App Engine跟踪两个相同模型类型的不同特性的排序，这就会给数据仓库中的模型类型的每次更新增加成本。所以，在开发者开始构建自定义索引之前，应仔细想想是否真的需要它们。在许多情况下，更好的方式是使用一个返回很多数据的简单的查询语句，然后在开发者的servlet代码中过滤这些数据，而不是使用一个复杂的查询语句，复杂查询语句的索引对每次更新会有负面的影响。这里有一个微妙的权衡：如果能够为开发者节省大量的通信，自定义索引就是有意义的。因此，如果开发者经常使用一个特定查询，并且可以大大减少由查询产生的数据量，那么，就可能值得使用自定义索引。但是，如果自定义索引并不经常被使用，或者在查询结果的数量上没有较大差异，那么开发者可能最好应简化处理，换句话说，坚持用标准索引。

为了进一步探讨，让我们来考虑一个真正需要自定义索引的例子。

我们需要做的是提出一个合理的、并不适合任何自动索引约束的查询。基本上，开发者需要自定义索引的地方，是当开发者要使用复合查询测试多个字段的相等性和单一字段的多种关系的时候。在我们的文件系统中，没有任何对象可以写出一个这样的查询！事实上，在许多App Engine程序中，开发者的数据模型最终都是类似于这样，没有情形需要自定义索引。但是，假设有一个稍微不同的模型，像这样：

```
class ResourceAttribute(db.Model):
    name = db.StringProperty(required=True)
    type = db.StringProperty(required=True)
    value = db.TextProperty(required=True)
```

因此，我们给ResourceAttribute模型增加了一个新特性，使用该特性可以给一个给定资源分配类型名称。类型名称将会告诉我们如何解析其值的特性。因此，举例来说，可以假想一个系统，其中的一些特性是一个安全系统的一部分。这些特性将采用security-level类型。那么，我们可能要找到所有名为protection，声明的安全级别在400到500间的security-level属性。

用GQL，会得到如下查询：

```
select a from attributes
    where a.name = "protection"
        and a.type = "security-level"
        and a.value >= "400" and a.value < 500
```

该查询不会有自动生成的索引，因为它有两种不同特性的等式测试，并且有对第三个特性的关系比较。App Engine的SDK不会自动生成索引，这是因为索引创建代价太大，所以它留给用户来决定是否真的需要该索引。

要创建自定义的索引，需要在我们的应用程序的app.yaml文件中声明该索引。app.yaml文件是Python应用程序的主配置文件。该文件用一种（丑陋的）标记语言编写，名为YAML。索引定义在该文件内的indexes节下边。每个索引的定义以被索引数据的类型开始。类型是模型类的名称。

然后，开发者需要声明一系列要索引的特性，以及它们是否应该按照升序或者降序排序。对于我们的查询，下面是一个索引的定义：

```
indexes:
- kind: ResourceAttribute
  properties:
  - name: name
  - name: type
  - name: value
    direction: asc
```

这些代码描述的是，我们希望索引ResourceAttribute，以便能够基于type和name特性进行等式比较，并且对value做不等式比较，通过索引将value特性按照升序排列。就是这样，一旦该索引在app.yaml文件中声明，并且部署了该应用程序，它会对已经在数据仓库中的所有数据生成该索引，并且在用户添加/或者更新数据的时候索引会被更新。

开发者还可以看到App Engine为正在运行的应用程序创建了什么索引。如果开发者进入App Engine的应用程序的控制面板，那么，有一个条目用来检查数据仓库的索引。例如，对于Java聊天室应用程序，PChatMessage上有一个索引，按日期和聊天的升序排列进行索引。

14.1.4 Java中的索引

在Java中，数据仓库索引几乎与其在Python中的工作方式完全相同，应用同样的规则和限制，并且两者非常相似。然而有一个区别。在Python中，当需要创建自定义索引时，我们通过在定义Python应用程序的app.yaml中添加表项来实现。但是，在App Engine中运行的Java应用程序并没有app.yaml文件。在Java中，开发者需要使用一个不同的文件，采用的是一个名为datastore-indices.xml的文件。正如开发者所看到的，这是一个XML文件。

我听到过很多人抱怨不得不使用像XML一样很复杂的内容，而不是Python中轻量级的YAML。就我个人而言，我更喜欢XML文件。YAML文件的语法相当古怪。我总是觉得很难保证其语法正确，可能很难知道什么时候可以添加另一个“-”，以及什么时候开始一个没有破折号的新行。我根本不想弄明白它的正确用法。XML可能有点丑陋，而且体积有可能会比必要代码多一倍。但是XML很清晰，很容易弄明白各部分的功能。

针对之前在Python中创建的索引，可以使用下面的XML为Java创建相同的索引：

filesystem/datastore-indexes.xml

```
<?xml version="1.0" encoding="utf-8"?>
<datastore-indexes
  autoGenerate="true">
    <datastore-index kind="ResourceAttribute" ancestor="false">
        <property name="name" direction="asc" />
        <property name="value" direction="desc" />
    </datastore-index>
</datastore-indexes>
```

14.2 更灵活的模型

数据仓库中发生的另一件重要的事情就是模型限制。我们到目前为止建立的模型已经承受了

很多的限制。类型的每个实例都必须有完全相同的特性，采用标准模型根本没有灵活性。但是有时候，标准模型限制性太强。当发生这种情况时，开发者有两个选择来解决该问题：expando模型和聚合模型（polymodels）。

expando模型是开发者可以在任何时候给模型实例任意添加新的特性的模型。它基本上是模型灵活性的极端情况。开发者可以在任何想要的时候，添加任何需要的内容。模型的所有实例，即开发者存储的所有的对象，需要是相同Python类的实例。

而在聚合模型中，开发者可以为模型实例定义一个超类。然后，开发者可以添加该类的子类。当进行查询时，会检索聚合模型基类以及聚合模型子类的所有实例。

开发者使用expando模型或聚合模型之前，请静下心来想一想。根据我的经验，大多数时候，如果开发者不能使用聚合模型或者expando模型简洁地定义其模型，那说明开发者还没有把事情想得足够清楚。有些情况下，开发者可能需要使用灵活的模型，但是的确需要时，这两个模型是现成可用的。对expando模型尤其如此：expando模型的极端灵活性有点像流程控制中`goto`语句的超级灵活性。这种灵活性比较适合解决某些问题，但在有些情况下，用它来解决虽然简单，却并不正确。

在采用聚合模型的情况下，内容非常清晰。开发者用与定义标准模型完全相同的方式定义一个模型类，只是该基类是从`polymodel.PolyModel`继承的。然后，开发者再按照通常的方式定义聚合模型的子类。

例如，我们可以以文件系统为例，使`ResourceAttribute`成为聚合模型添加类型属性。然后，就可以为字符串属性、整数属性等添加`ResourceAttribute`的子类：

filesystem/poly.py

```
class ResourceAttribute(polymodel.PolyModel):
    name = db.StringProperty(required=True)

class StringAttribute(ResourceAttribute):
    strVal = db.StringProperty(required=True)

class IntegerAttribute(ResourceAttribute):
    intVal = db.IntegerProperty(required=True)
```

如果真的必须要用expando模型，也是非常简单的，从`google.appengine.ext.db.Expando`继承类即可。在expando模型中，开发者根本不需要定义任何特性。当然也可以定义一组固定特性，这些特性会被包含在该模型的每个实例中，但同时，Python对象的各个字段也将隐式地被视为无类型特性。采用这种方式时，也就失去了数据仓库通常所提供的所有帮助功能。开发者不会得到自动引用，也不会得到任何数据验证。Expando模型是完全灵活、完全非结构化的，人们可能不希望使用它。我曾看到过一个expando模型的例子，确实是需要使用expando。我不想在这里向读者展示这样的例子，因为那毫无意义。

或许可以定义一个使用expando模型的文件系统版本。我能完全消除`ResourceAttribute`，而只使用Python资源对象的字段来存储特性。但这是一个糟糕的解决方案，失去了对属性定义查

询的能力，以及很多结构一致性。能获得什么呢？坦白地说，没有多少。对于更新属性而言，我能够获得稍微好一点的语法，收获仅限于此。

14.3　事务、关键字和实体组

回顾10.2节，我们谈到了事务。它们为何如此重要？它们在Java中又是怎样自动工作呢？原来，我们在Python中也一直在用事务，只是没有费心去了解它们而已。与使用Python的App Engine中的许多其他工作一样，默认行为已经设置好了，而且大部分时候，默认设置就是正确的。但像往常一样，有些情况下开发者想要进行的并不是缺省内容，因此需要了解事务究竟是如何工作的。

Python中的事务是由实体组（entity group）驱动的。开发者在Python中创建的每一个持久性对象，都是称为实体组的对象组的一部分。每次开发者从Python中往数据仓库存储一个更改时，对同一个实体组的对象的所有更改将在一个事务中保存。因此，如果开发者可以将其所要存储的所有内容安排在一个实体组中，就不需要担心事务了，它将会正常工作。

实体组使用祖先关系进行管理，每个对象都自动与它的父对象在相同的实体组中。开发者在创建对象时创建父子关系，当创建一个持久性Python对象时，可以通过使用一个命名的`parent`参数指定该对象的父对象。如果不指定父对象，那么该对象是根，它定义了它自己的实体组。

如果开发者不需要使用一个对象集合内的事务行为，那么最好就让对象作为根。事务非常有用，但代价并不小。以数据仓库和App Engine分布式环境来看，实体组不能是分布式的。实体组需要集中位于一台服务器上。这意味着，当开发者扩展到大量的用户和对象时，将太多的东西放到同一个实体组内可能使其应用程序的执行变得非常糟糕。

例如，在聊天应用程序中，我们可以将所有的聊天消息放入一个实体组中。这样做会有一定的优势，如提供非常强有力的一致性保证。但随着时间的推移，我们很容易地就会在数百个聊天室中拥有数万或数十万个聊天消息，并且在事务进行过程中，向任何聊天室添加消息都会阻止其他聊天室用户的更新。

另一方面，我们也不想忽略事务的价值。在文件系统中，如果用户更新一个资源的一组属性，然后保存，用户必然期望更新或者成功，或者失败。所以，在我们的文件系统资源的例子中，使属性成为资源的子节点是有意义的，从而使属性更新成为它们的父对象的事务的一部分。要实现这些功能，我们唯一需要做的修改是`Resource`的`SetAttribute`方法：

filesystem/filesystem_servlet.py

```
def SetAttribute(self, name, value):
    if name == "children":
        self.children = [ de.key() for de in value ]
        self.put()
    else:
        for attr_key in self.attributes:
            attr = ResourceAttributeModel.get(attr_key)
            if attr.name == name:
                attr.value = value
```

```
            attr.put()
            return
        newAttr = ResourceAttribute(parent=self, name=name, value=value)
        newAttr.put()
        self.attributes.append(newAttr.key())
        self.put()
```

这个代码与我们以前见过的setAttribute方法之间的唯一区别是创建新的ResourceAttribute对象的调用。在这个新版本中，我们为ResourceAttribute提供了一个parent参数，将属性放到与Resource对象本身相同的资源组中。

除了事务，资源组也影响了对象关键字。关键字编码了资源路径的信息，即从根对象到特定对象的绝对路径。这个特点可以用在查询中，开发者可以借此缩小查询范围，使查询中只包括特定对象作为祖先的对象。通过在查询中使用一个特殊的查询子句ANCESTOR IS可以实现此功能。例如，在文件系统中，属性有资源作为父节点，从而，我们可以查询一个特定资源的属性值。如果想要资源R的length属性，我们可以写一个类似下面的GQL查询：

```
query = db.GqlQuery("SELECT * FROM ResourceAttribute WHERE "
"name = :name AND ANCESTOR IS :parent",
name="length", parent=R.key())
```

14.4　策略和一致性模型

我们需要学习的有关数据仓库的最后一个内容涉及策略（policy）这个概念。策略是描述定义系统该如何执行的高层次选项的总称。在数据仓库中，哪些索引集会自动生成就是一个策略问题。有两个主要策略问题会影响开发者可以控制的数据仓库查询。

- 一致性模型：系统处理在App Engine云中运行的应用程序的不同实例之间的一致性方法。毫无疑问，一致性策略是最关键的策略问题。我们后面将介绍更多的内容。
- 截止期限：查询失败之前应该允许耗费的最大时间。有时候，根本就等不起某些内容——如果在特定时段内没有产生响应，就根本没有生成的必要了。例如，在聊天应用程序中，如果响应更新查询的时间超过了1秒钟，就没有响应的机会了，因为另一个更新请求已经被放入队列中——如果每个查询时间超过两次查询间的时间，就会造成不断增长的查询积压。让查询失败并返回，比生成一个不断增长的积压要好得多！截止期限表达的是："如果这花费了太长时间，那就让它失败，而不是等待。"

如上所述，最重要的策略问题涉及一致性模型。默认情况下，数据仓库采用所谓的强一致性模型。强一致性是指进行相同查询的任何两个客户端，总是保证得到相同的结果。这似乎是一个显而易见的需求：如果一个给定的查询并不总是产生相同的结果，一定有错误，不对吗？

不一定。如果数据是静态的，那么任何查询会得到一个精确的、定义良好的结果，并且每一个查询总是返回该结果。但是，更新会使其复杂化。结果的定义需要考虑特定的时间。然而，正如我们讨论过的，分布式环境中的时间有些棘手。

想象如下的场景。

(1) 客户端1获取资源R1，并将属性A设置为字符串abc。

(2) 客户端2获取资源R1。

(3) 客户端1将R1的属性A更新为字符串def。

(4) 客户端2获取R1的属性A。

客户端2获得的R1的属性应该是什么值呢?

根据强一致性，它应该得到def。属性被更新了，客户端1看到了更新，因此所有其他客户端也必须看到更新。对于许多应用程序，这正是开发者想要的结果。例如，如果你正在运行网上商店，并告诉用户还剩一个商品，你不想将此商品出售两次：一旦任何人购买了该商品，现在该商品库存为零的事实应立即对所有人可见。强一致性保证了这点。用行业术语说，强一致性是保守的，它确保总是向每个人提供正确的结果，不管成本多大。

但强一致性代价确实很高。在我们的例子中，客户端2在客户端1进行更新之前进行了资源的完整取回操作。如果客户端2要看到属性的更新值，它需要给服务器发送新的查询，通过引用取回属性。客户端2不能一次就获取所有内容。这一轮行程不是免费的，有显著的成本。有时候，这样的花费是不值得付出的。

还有另一种一致性政策，称为最终一致性，代价会低一些。在最终一致性中，发生的情况是，最终，各部分将对数据的值达成一致。但也有可能在一个短的时间内，各部分数据并不一致。换句话说，这是一种模糊一致性：每当用户做一个更新时，有一个模糊的时期，有人可能会看到旧的、未更新的值。

对于数据库领域，最终一致性被称为BASE一致性(Basically Available，Soft State，with Eventual consistency)，对应ACID（Atom，Consistent，Isolated，Durable state）。

最终一致性对很多工作而言都很好。例如，在聊天室例子中，最终一致性可能意味着两个不同的客户端在最后一条消息发布时不一致。但是由于客户端1秒通过查询检查两次，所以他们的不一致时间绝不会超过半秒！对于聊天室而言，这完全是可接受的。

同样，对于文件系统，人们普遍认可如果两个程序试图在同一时间修改同一个文件，结果可能会出错。如果开发者使用最终一致性，可能在涉及属性值等的地方获得竞争条件，但是，有多个客户端并发更新属性值，难免会出现竞争条件。强一致性减慢了速度，并消除了一些竞争，但并不是全部竞争。

所以，假设我们要在查询中使用最终一致性，该查询返回的结果也许稍有过时，但我们明白并且对此可以接受。我们如何真正使用最终一致性呢?

事实上，必须一直分析到数据仓库接口的底层，才能弄清该问题。在底层，数据仓库使用Google的异步RPC系统，如同在第12章中所见到的那样，我们需要修改查询中使用的RPC的内部参数。策略问题是由RPC对象决定的，因此，我们要改变策略，就需要改变RPC对象。

查询操作Query的fetch方法，其实需要一个名为rpc的参数。它用默认的参数值定义，能够用标准参数创建了一个新的RPC对象。当开发者想改变策略的功能时，需要用正确的策略参数创建自己的RPC对象。

开发者可以通过调用db.create_rpc创建RPC对象。该函数需要两个命名参数：read_policy

和deadline。要使用最终一致性，请设置read_policy = db.EVENTUAL_CONSISTENCY。开发者不需要声明强一致性，但如果真的想要用强一致性，那么可以使用read_policy = db.STRONG_CONSISTENCY显式地设置默认值。对于deadline，该参数的值表示在查询失败之前想要等待的最低秒数。截止时间过后，并不保证它会立即失败，唯一可以保证的是不会在截止时间前失败。

开发者创建好一个RPC对象之后，可以将它作为查询的参数。因此，举例来说，如果q是开发者的查询，那么可以用此代码实现使用截止时间为2秒的最终一致性：

```
q = ...
my_rpc = db.create_rpc(deadline=2, read_policy=db.EVENTUAL_CONSISTENCY)
result = q.fetch(rpc=my_rpc)
```

同样，如果开发者使用Python迭代器访问查询的结果，那么可以在run方法中提供RPC对象，而不是fetch方法中：

```
q = ...
my_rpc = db.create_rpc(deadline=2, read_policy=db.EVENTUAL_CONSISTENCY)
for r in q.run(rpc=my_rpc):
…
```

当然，正如在许多高级功能中一样：仅仅因为你可以这样使用，并不意味着你应该这样做。大多数时候，开发者最好进行fetch操作，因为fetch更高效。

在Java中的过程是相似的，但开发者不用显式地使用RPC对象。相反，App Engine使用的JDO API提供了对查询对象直接设置策略的方法。该方法并不是创建一个由策略设置的RPC对象，然后将其传递给查询方法，而是开发者可以直接在查询对象上设置策略：

```
Query q = persister.newQuery(ChatMessage.class);
q.addExtension("datanucleus.appengine.datastoreReadConsistency", "EVENTUAL");
q.setTimeoutMillis(2000);
```

14.5　渐进式取回

默认情况下，开发者在数据仓库中做查询时，一次返回该查询的所有结果。在性能方面，这通常是最好的选择：这样可能在大多数情况下提供最佳的运行时性能。但有时，一个查询会返回大量数据，难以一次在内存中保存。所以，开发者需要将查询策略改变为以组块为单位访问数据。

查询的工作方式是全部获取还是渐进式（即一个组块接着一个组块）获取，确实是个策略问题。但坦率来说，它要比我们所见的一致性策略获得的支持更好。各种不同的一致性模型的支持非常草率。App Engine的设计者很自然地认为强一致性几乎总是正确的。于是，他们特意让最终一致性使用起来显得有些笨拙。开发者需要三思自己确实想要使用最终一致性，因为使用最终一致性非常痛苦。但渐进式查询呢？这是非常直观的功能，因此渐进式查询功能的支持更稳定。

对于渐进式查询，需要使用游标（cursor）。由开发者进行查询，但是查询时，会对查询应该返回的结果数量有所限制。然后，开发者可以对内容进行正常迭代。因为做了一个有限制的取回操作，所以，开发者现在会有一个游标，指向其已经取回的结果后边的第一个查询结果的位置。

要进行有限制的查询，开发者只要向查询语句提供一个数字参数即可。例如，要对所有资源

对象做查询，但每次只取回50个，可以使用有限制的查询：

```
q = Resource.all()
batch = q.fetch(50)
```

然后，开发者完成首批迭代后，就可以通过调用`q.cursor()`得到该查询的游标。

对于下一批查询，接下来开发者可以按照如下代码设置游标，并使用该游标：

```
q = Resource.all()
batch = q.fetch(50)
c = q.cursor()
...
q.with_cursor(c)
q.fetch(50)
```

每一批查询之后，开发者都可以从查询对象上获取游标。然后，在做下一个查询时，就可以向其提供游标，作为查询的起点。事实上，游标本身能够被用作持久性对象：开发者可以保存一个游标，然后在以后的查询中使用。（在下一章中学习Memcache服务时，这将变得特别有用。）

在过去的两章中，我们已经了解了App Engine数据仓库中的一些更高级的功能以及局限性。不过，这些还只是表面文章。数据仓库在不断发展，将来会提供更多更好的功能，所以开发者一定要时常查看App Engine的文档。但其主要功能和关键的设计要点，都会保持不变。

正如我们所看到的，数据仓库的真正秘密是要能理解它所采用的基本模式。这是一个用于存储对象的系统，一切其他事情——从事务系统到查询语言，到关键结构，到引用——都是从基本的原则派生而来的。数据仓库就是有关如何存储持久性对象的——存储的不是表、不是集合、不是列表，也不是树。

App Engine数据仓库是一个灵活的、轻量级的、基于对象的持久性机制，而不是一个关系数据库。如果开发者记住了这一点，就会发现数据仓库通常是以很直观的方式运行的。

第 15 章

Google App Engine服务

15

现代软件工程的主要关注点之一是重用。我们不希望不断地重新开发同样的东西。对于最常见的软件组件，我们创建包含组件实现的库，然后只要使用这些库即可，而不用再编写自己的实现代码。例如，在C++中，并不会在每次构建用到哈希表的应用程序时都编写一个新的哈希表实现，我们只需使用标准模板库中的映射类型即可。在Python应用程序中建立用户界面时，并不是从编写用单个像素绘制按钮的代码开始，而是找到一个像`wxWindows`这样的库，然后使用已有的库。

云中做法也几乎相同，只是不使用库，而是使用服务。库和服务之间的区别是微妙而重要的。库（library）是静态的，它是计算机上拥有的一堆代码，我们可以在应用程序里包含库。使用库时，库的代码成为我们的应用程序的一部分。在云的世界中，服务（service）是在云中某处运行的，它本质上是应用程序本身，但是，该应用程序的运行是为了向其他应用程序提供某种能力。

服务对于我们而言并不是全新的。在之前的几章中，我们一直在详细地了解数据仓库，而数据仓库就是App Engine提供的服务之一。在第5章中，我们还看到了另一个用于管理用户登录的App Engine服务。

我们有很多标准桌面开发的库，可以提供给应用程序其可能需要的不同功能的实现，同样，云中（特别是在App Engine中）有很多服务，可提供给云应用程序所需的各种功能。本章将讨论App Engine提供的通用服务的集合，但不会巨细靡遗地加以介绍，因为用于App Engine开发的服务有很多，而且Google的App Engine团队还在不断地扩大他们所提供的服务集。我们将列举一些服务的例子，从而让读者进一步了解服务所提供的功能以及服务接口的工作原理。

本章稍显芜杂。App Engine提供了大量服务，而且其中大部分服务都很简单，而且也非常杂乱，不需要花费一整章对每一服务进行细致的介绍。读者可能会在需要之时才从本章中抽取相应的服务。

15.1　快速访问重要内容：Memcache 服务

Memcache可能是开发者在实际的应用程序中使用最多的服务之一——它非常简单，只需要一两段话就能说清。创建的每个App Engine应用程序都会将数据存储在数据仓库中。但问题是，从数据仓库取回东西需要时间。当你第一次考虑取回操作所花费的时间时，似乎微不足道——通

常只有不到一秒钟而已。但如果设想一下有可能一秒钟要提供数千个请求的真实服务时，这些取回操作时间加起来就会变得不容忽视。

对于Memcache的名称，开发者丝毫不会惊讶，该名称正表示其所提供的功能：内存中的缓存。开发者会一直在数据仓库中保存其数据的主副本。但同时也会在Memcache中存放一个副本，而且只要数据副本存在于Memcache中，取回操作实际上就是即时的。

Memcache是简单服务的一个很好的例子。它通过一个简单的库API提供其功能。在后台，它实现了一些很微妙的工作，从而提供一个高速缓存，能够从App Engine云中的机器进行访问，但是，所有这些对于作为客户端的用户而言，都是不可见的。该服务有一组功能，只有开发者可以调用，看起来就像一个简单的内存缓存。这是一个非常优雅且十分有用的小服务。

当然，Memcache还有一些要注意的地方。首先，它是易失性存储，无法保证开发者已存储在Memcache中的内容稍后还会存在。Memcache能让操作变快，但不能替代如数据仓库那样的非易失性存储。如果开发者一段时间内不访问高速缓存中的对象，或者开发者放在缓存中的东西太多了，用完了Memcache的空间，就会开始丢弃缓存的内容。每当开发者使用Memcache时，都需要包含一个对数据仓库的回调函数。不要以为只要在Memcache中放入内容，你就可以自动取回该内容！

其次，Memcache是用于存放小东西的。开发者发送到Memcache的任何请求最大为1兆字节。所以，开发者不能存储大于1兆字节的内容，如果要用一个单一的请求（无论是get还是put）访问多个对象，它们的总大小也不能超过1兆字节。

因此，举例来说，如果想在我们的聊天应用程序中使用Memcache，就不能用它实现对所有聊天消息列表的快速访问，聊天消息列表很容易会增长到1兆字节以上。但是，可以用它来存储所有可用的聊天室列表：按照我们的设置方式，聊天室列表永远不会大于1兆字节。

15.1.1 在Python中使用Memcache

在Python中，Memcache基本上是一个简单的包含键/值对的字典。开发者可以用一个以字符串作值的键把对象放到其中。在Python中，可以采用类似下面的代码在缓存中存储对象：

```
from google.appengine.api import memcache

memcache.set(key="aString" value=aValue)
```

而且，取回操作也很简单：

```
v = memcache.get(key="aString")
```

函数get会返回相应的值，或者，如果关键字不在缓存中，则返回None。

开发者也可以在一个调用中进行多个更新/取回。要设置多个值，使用set_multi，并传递给该函数一个字典作为参数：

```
memcache.set_multi({ "key1": "value1", "key2": "value2"})
```

要取回多个值，使用get_multi，并传递给该函数一个键列表。该函数返回包含键/值对的字典：

```
dict = memcache.get_multi(["key1", "key2"])
```

最后，正如我前面提到的，如果Memcache一段时间不被访问，或者它的空间用完，Memcache会丢弃内容。开发者可以通过`time`参数给Memcache提示，告知它应该等多久才开始丢弃内容。这个时间参数的意义基本上等于告诉它“请在内存中最多保留该内容这么长时间”。当然，Memcache无法保证会将数据保留那么长时间如果缓存在此时间之前用完了可用空间，仍然会丢弃该值。但Memcache不会因为该内容太久没有被使用，在指定时间过期之前就将其丢弃。该时间参数是以秒数来计算的，因此，如果要将内容存储至少两个小时，可以使用`memcache.set(key="foo", value="bar", time=7200)`。

15.1.2　在Java中使用Memcache

在平常的App Engine风格中，涉及到提供给Java的Memcache访问功能时，App Engine是基于标准的Java API构建的。有一个在Java中使用高速缓存的建议标准机制，可参见Java标准请求文档JSR107的描述。App Engine使用JSR107的API提供了从Java 对Memcache的访问。同样，作为标准过程的一部分而设计好的API比定制化设计更为繁琐。因此，从Java中使用Memcache需要更多的样板。但从概念上说，Java中的Memcache与我们在Python中看到的仍然相同。事实上，两者必须相同，因为它们有着相同的底层实现。虽然表面API不同，但服务却完全一样。高速缓存就是从键到值的映射。在Java中，值需要被序列化，这正是开发者要把数据存放到数据仓库时所需要的方式。在此之前，我们看到，在Python中要把内容放到Memcache中，可以只进行`memcache.set`。而在Java中，使用样板，代码可能如下：

```
import java.util.Map;
import net.sf.jsr107.Cache;
import net.sf.jsr107.CacheException;
import net.sf.jsr107.CacheFactory;
import net.sf.jsr107.CacheManager;
import com.google.appengine.api.memcache.stdimpl.GCacheFactory;

class MyApplicationClass {
    Cache cache;
    Map cacheConfig = new HashMap();
    {
        try {
            CacheFactory factory = CacheManager.getInstance().getCacheFactory();
            cache = factory.createCache(cacheConfig);
         } catch (CacheException e) {
            // 不用担心
         }
    }
    cache.put(aKey, aValue);
    // ...
}
```

是的，与Python版本相比，这段代码有点冗长。而且，由于使用了无类型映射，该代码会在编译时产生一个警告。此外，还使用了令人担忧的空`catch`子句。不过别害怕。`Map`参数采用了一套特殊实现的键/值二元组，然而，它刻意使类型未作具体声明，以便实现过程中可以使用其想要

的任何关键字和值。在App Engine中，开发者总会得到一个缓存实例：CacheException永远不会被CacheFactory触发，所以开发者可以保留空的catch块。

只要开发者觉得样板不碍事，它就变得像在Python中一样简单。要存储一个对象，使用cache.put，而要取回对象，使用cache.get：

```
cache.put("aKey", aSerializableValue);

value = (ValueType)cache.get(aSerializableValue);
```

此外，还有的getAll和putAll方法，其使用就像Python中的get_multi和put_multi。getAll方法需要一个键的集合，返回键/值二元组的映射集合；putAll需要键/值的二元组映射，并更新所有内容。

不幸的是，开发者用来从Java访问Memcache的JSR107标准，不允许对单个对象设置缓存过期时间。在Java中，这是高速缓存的一个策略特性。当开发者得到缓存时，可以提供配置参数，其中之一就是缓存过期时间。因此，如果要求所有缓存对象保留至少两个小时左右，在创建缓存时，就需要给特性映射添加一个参数：

```
class MyApplicationClass {
    Cache cache;
    Map cacheConfig = new HashMap();
    cacheConfig.put(GCacheFactory.EXPIRATION_DELTA, 3600);
    {
        try {
          CacheFactory factory = CacheManager.getInstance().getCacheFactory();
          cache = factory.createCache(cacheConfig);
        } catch (CacheException e) {
          // 不用担心
        }
    }
    // 类的剩余部分
}
```

15.1.3 应该缓存何种内容

Memcache是针对少量频繁访问内容的易失性存储系统。不要只想把一切内容自动都转存到缓存中，必须谨慎且有选择地去做，只有当真正会不断访问某些内容时，才应将其放入缓存。

例如，在文件系统应用程序中，每个请求都需要获取根目录。请求到来时总会包含我们想要获取或者更新的资源的路径名，为了找到该资源，先从根目录资源开始，使用路径名查找特定资源(也即请求目标)。我们会不断用到根目录资源，因而该内容应在Memcache中。而另一方面，我们获取的一个典型资源可能不该在Memcache中。在这两者之间，正是事情的巧妙之处：层次结构中低层的目录将会被比较频繁地访问；越到层次结构中的较高层，它们可能被取回的频率就越小。在文件系统的例子中，高层的目录可能并不需要被缓存。

正如之前所提到的那样，在聊天应用程序例子中，将所有聊天消息放到缓存中毫无意义，但将可用聊天室列表保留在缓存中应该是不错的。聊天室对象很小，而且它们会被不断访问。

15.1.4　缓存访问模式

我们之前看到了如何从Memcache中取回东西。但是，因为Memcache是易失性的，要查找的值可能不在其中，所以在实际应用中，当我们使用Memcache时，总是使用一种模式取回值：首先是尝试缓存，如果找不到该对象，就回到数据仓库中查询。

在文件系统中，我们实际上总是取回文件系统对象。根目录资源有一个引用特性，所以看起来并不像是我们使用一个查询语句获取了根目录。真正的查询看起来像是针对文件系统对象的，然后通过访问该文件系统对象的一个特性来获取根目录。但要记住，引用特性实际上是自动查询。因此，通过缓存根目录，我们可以消除两个查询！

在原始版本的`FilesystemResourceHandler.get`中，我们通过调用`getFilesystem`获取根对象，该函数会进行一个GQL查询取回文件系统对象，然后我们再从文件系统中取回根对象。在Memcache版本中，代码改变为以下内容：

filesystem/cachedfilesystemservlet.py

```
root = memcache.get(key="root")
if root is None:
    query = Filesystem.gql("")
    filesystem = query.get()
    root = filesystem.getRoot()
    memcache.put("root", root)
```

这正是开发者会一次又一次地使用Memcache的模式。首先，尝试从缓存中取回该对象；如果它不存在，则进行一个数据仓库查询以取回该对象，然后将取回的对象放入缓存。接着就可以继续使用该对象了。因此，如果对象在高速缓存中，开发者即刻就可以获取；如果它不在缓存中，需要做一个查询并取回对象，然后把对象放到缓存中，以便在下一次需要时可立即获取到。

15.2　访问其他内容：URL Fetch 服务

网络服务和应用程序是通过HTTP进行交互的。如果开发者想编写一个与其他网络服务进行交互的应用程序，就需要编写发送HTTP查询的代码。由于这是很常做的工作，App Engine提供了一个服务，不用开发者手工编写HTTP协议交互。

事实上，Java和Python都提供了一套进行HTTP交互的标准库。在Python中，有三种不同的库：urllib、urllib2和httplib。Java（像往常一样）更加严格，它有一个标准的HTTP库`java.net.URLConnection`。开发者可以按照其在标准的Python或Java中的方式，在App Engine中使用任何一个库，但是，在后台，App Engine将库的实现替换为在App Engine环境中优化过的、既高效又可扩展的定制化版本。

因此，如果我们的文件系统服务在`mcfile.appspot.com`运行，另一个Python应用程序可以使用下面的代码取回资源`markcc/book.html`：

```
import urllib
```

```
result = urlfetch.fetch("http://mcfile.appspot.com/markcc/book.html")
if result.status_code == 200:
    # 成功提取文件内容到result.content
```

要知道，像云应用程序的其他所有内容一样，开发者要对所使用的资源付费。在App Engine中，开发者要为其使用的带宽付费，所以不要做得太过。使用App Engine编写某些程序，如网络爬虫，成本可并不低。

15.3 与人沟通：Mail 和 Chat 服务

除了HTTP，App Engine应用程序或服务还可用其他协议来跟其他程序沟通。例如，应用程序可能会用即时消息向用户发送警报。虽然并不常见，但个别情况确实需要如此。

App Engine提供了一个能让应用程序通过即时消息与用户交互的服务。该服务非常容易，开发者不必担心究竟要使用什么样的即时消息。经过多年的发展，各种聊天服务已经商定了一个标准的描述即时消息的协议，称为XMPP（eXtensible Messaging and Presence Protocol，可扩展通讯和显现协议）。App Engine提供的服务使开发者可以使用XMPP发送和接收聊天消息。App Engine对所有的XMPP服务都提供一个标准的接口：通过使用App Engine的XMPP服务，开发者可以与几乎任意使用该标准的聊天服务的用户沟通——AIM、Google Talk，MSN Chat，等等。（在后台，雅虎和MSN Chat使用专有协议，但是，它们都有XMPP网关，这意味着，如果开发者愿意建立与网关的连接，就能与它们交谈。）

15

XMPP服务的设置方式是非常典型的App Engine处理通信服务的方式。各种服务使用相同的模式让开发者发送和接收电子邮件、发送和接收存储的大块数据、接收排队任务的通知，以及进行其他工作。

为了发送邮件，XMPP提供了一个对象，可以让开发者使用方法调用执行发送。这差不多是目前为止我们看到的使用了几乎所有服务的接口类型。为了接收消息，XMPP提供了一个HTTP门面。也就是说，它得到传入的消息，并作为一个特定URL的HTTP请求呈现给开发者。为了处理收到的消息，开发者只需要为该服务的传入URL实现一个标准的HTTP消息处理程序，或者servlet即可。这看起来似乎有点奇怪——为什么要获取一个即时消息，并将它变成一个HTTP的POST请求呢？但这样做的关键是，开发者只需要知道如何编写一种处理程序。在App Engine中，开发者编写的处理传入数据的所有内容都是HTTP的`PUT`或者`POST`处理程序。这样，开发者只需要知道一个接口，只需要关心一种处理程序。App Engine称这种功能为*网络钩子*（webhook）：在App Engine中，与各种协议和服务交互的能力都通过webhooks提供，就彷佛提供了一座桥梁，将采用其他协议的请求呈现为好像使用的是HTTP请求一样。webhooks是App Engine最出色的功能之一，开发者可以与任何服务交互，而无需知道其采用的协议的细节——只需要编写HTTP的servlet即可。

15.3.1 发送聊天消息

实际上，开发者只需知道两种方法即可，一个用于检查用户是否登录，一个用于发送消息。

要检查用户是否登录，可使用xmpp.get_presence。因此举例来说，要检查我是否登录，然后发送给我一个消息，可以做如下工作：

```
from google.appengine.api import xmpp
...
if xmpp.get_presence("markcc@gmail.com"):
    status_code = xmpp.send_message("markcc@gmail.com", "Hi Mark!")
    if status_code != xmpp.NO_ERROR:
        self.response.setStatus(500, "Sorry, couldn't send a message to Mark")
```

或者，在Java中：

```
import com.google.appengine.api.xmpp.JID;
import com.google.appengine.api.xmpp.Message;
import com.google.appengine.api.xmpp.MessageBuilder;
import com.google.appengine.api.xmpp.SendResponse;
import com.google.appengine.api.xmpp.XMPPService;
import com.google.appengine.api.xmpp.XMPPServiceFactory;

class JavaXMPPExample {

    public sendMessage() {
        // 获取一个表示用户的标识符对象
        JID jid = new JID("example@gmail.com");
        // 构建消息
        Message msg = new MessageBuilder()
                .withRecipientJids(jid)
                .withBody("Hi Mark!)
                .build();
        boolean messageSent = false;
        // 获取XMPP服务
        XMPPService xmpp = XMPPServiceFactory.getXMPPService();
        // 检查用户是否登录
        if (xmpp.getPresence(jid).isAvailable()) {
          //发送消息
          SendResponse status = xmpp.sendMessage(msg);
          if ((status.getStatusMap().get(jid) != SendResponse.Status.SUCCESS)) {
          }
        }
    }
}
```

像往常一样，Java比较复杂一点，需要为用户创建一个标识对象，而不仅仅是直接传递一个字符串。但同样地，这只是样板，编写一次，然后就可以重用。

15.3.2　接收即时消息

从XMPP接收消息比发送稍微难一些，但其实也非常简单。对于App Engine代码而言，XMPP只不过是一种使用HTTP的程式化方式。使用App Engine的webhook，看起来好像都是在HTTP之上以POST形式发送消息实现的。无需担心XMPP到底如何工作，App Engine已为开发者处理好了所有事情。开发者需要担心的就是如何处理App Engine发送给开发者的HTTP请求。因此，要想

让应用程序实现从XMPP服务中接收XMPP消息，只需要做以下三件事情。

(1) 告诉App Engine你想要接收XMPP消息。开发者通过在其应用程序配置文件中添加一个表项完成该工作——在Python中，该配置文件是`web.xml`，而在Java中，该配置文件是`appengine-web.xml`。

(2) 编写一个XMPP处理程序，这确实只是一个实现`POST`的标准的HTTP请求处理程序。

(3) 注册XMPP处理程序。App Engine总是将XMPP消息映射到一个特定的URL：`/_ah/xmpp/message/chat/`，所以，开发者需要将XMPP处理程序注册为该网址的处理程序。

有了上述的设置，人们就可以通过给`anything@your-app-id.appspot.com`发送一条消息来向开发者的应用程序发送聊天消息。

15.3.3 在Python中处理聊天消息

要在Python中接收XMPP消息，开发者首先需要在`app.yaml`中加入以下表项：

```
inbound_services:
- xmpp_message
```

然后，编写一个XMPP处理程序：

```
class XMPPHandler(webapp.RequestHandler):
    def post(self):
        message = xmpp.Message(self.request.POST)
        message.reply("Received message: %s\nHello to you too" % message.body)
```

开发者只需要获取`POST`请求，让XMPP服务将其转换为一个XMPP消息对象，然后得到消息（`message.body`）、发送者（`message.sender`）或预期的收件人（`message.to`）即可。

最后，开发者需要注册该处理程序：

```
application = webapp.WSGIApplication([('/_ah/xmpp/message/chat/', XMPPHandler)])
```

15.3.4 在Java中接收聊天消息

Java中的基本步骤与Python中的几乎相同，但是，像往常一样，为了匹配语言间的差异，其形式略有不同。首先，开发者通过在应用程序的配置文件`appengine-web.xml`中添加一个表项，告诉App Engine开发者要接收聊天消息。

```
<inbound-services>
    <service>xmpp_message</service>
</inbound-services>
```

然后，编写一个消息处理程序：

```
import java.io.IOException;
import javax.servlet.http.*;
import com.google.appengine.api.xmpp.JID;
import com.google.appengine.api.xmpp.Message;
import com.google.appengine.api.xmpp.XMPPService;
import com.google.appengine.api.xmpp.XMPPServiceFactory;
```

15

```
public class XMPPReceiverServlet extends HttpServlet {
    public void doPost(HttpServletRequest req, HttpServletResponse res)
          throws IOException {
          XMPPService xmpp = XMPPServiceFactory.getXMPPService();
          Message message = xmpp.parseMessage(req);

          JID fromJid = message.getFromJid();
          String body = message.getBody();
          // ...
    }
}
```

最后，通过在开发者的web.xml文件中加入servlet和servlet-mapping表项，注册此处理程序，如以下配置所示：

```
<servlet>
    <servlet-name>xmppreceiver</servlet-name>
    <servlet-class>XMPPReceiverServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>xmppreceiver</servlet-name>
    <url-pattern>/_ah/xmpp/message/chat/</url-pattern>
</servlet-mapping>
```

15.4　发送和接收电子邮件

与XMPP的机制类似，开发者还可以在App Engine中发送和接收电子邮件。虽然起初这项功能看起来有点傻，但如果仔细想一下，其实你就会发现，发送邮件实际上比发送即时消息更为普遍，更为有用。

电子邮件常用于在线注册、通知，并进行验证。当用户第一次使用一个提供聊天服务的网站时，通常必须注册，而且用户要通过电子邮件注册。如果开发者有一家商店，通常会通过电子邮件发送收据和确认信息。如果用户需要重新设置密码，开发者通常会通过电子邮件完成这项工作。如果用户在有内容更改时想知道相关情况，比如说当他们没有登录的时候是否有人在聊天室向某个讨论添加了内容，或者商店里缺货的商品有货了，用户往往会希望通过电子邮件了解这些信息。

所以说，发送电子邮件是很常见的。它既不像持久性数据那样无处不在，也不像发送即时消息那样不常见。App Engine提供了一个非常简单灵活的服务，让开发者的应用程序能够发送和接收电子邮件。

15.4.1　发送邮件

发送邮件再简单不过了。在Python中，一旦开发者导入了邮件服务，发送邮件就是一行代码。要给我从chat-service@markcc-java-chat.appspot.com发送主题为“Hi Mark!”，消息主题为“I am still running”的邮件，开发者需要做的所有工作就是如下这些：

```
from google.appengine.api import mail

mail.send_mail("chat-service@markcc-java-chat.appspot.com",
    "markcc@gmail.com", "Hi Mark!", "I'm still running")
```

在Java中更复杂一点。Java在javax.mail包中为发送邮件提供了一个标准的API，而App Engine只是提供了一个实现。与上述Python脚本中的简单的一行代码等价的Java代码是这样的：

```
import java.util.Properties;

import javax.mail.Message;
import javax.mail.MessagingException;
import javax.mail.Session;
import javax.mail.Transport;
import javax.mail.internet.AddressException;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeMessage;

...

    Session mailSession = Session.getDefaultInstance(new Properties(), null);
    try {
        Message msg = new MimeMessage(session);
        msg.setFrom(new InternetAddress("chat-service@markcc-java-chat.appspot.com"));
        msg.addRecipient(Message.RecipientType.TO,
            new InternetAddress("markcc@gmail.com"));
        msg.setSubject("Hi Mark!");
        msg.setText("I'm still running!");
        Transport.send(msg);
    } catch (AddressException e) {
        // 邮件地址错误时的处理方式
    } catch (MessagingException e) {
        // 发送失败（非邮件地址出错）时的处理方式
    }
```

15

15.4.2 接收邮件

相对来说，接收邮件稍微复杂一些，它遵循通用的App Engine通过服务接收数据的模式。开发者通过在应用程序的配置中添加表项，告诉App Engine想要从该服务接收数据。然后App Engine会将从该服务传入的消息路由到一个虚拟的HTTP的URL。在这个网址，消息能够被开发者的消息处理程序或者servlet处理。为简单起见，我将给读者展示一个Python应用程序如何能够接收电子邮件的简单例子。像往常一样，Java的解决方案与Python几乎相同，但有一点更复杂。

该模式的第一步是告诉App Engine，开发者的应用程序要使用电子邮件服务接收传入的电子邮件。要实现此功能，开发者只需要在其app.yaml文件添加一个inbound_services表项：

```
inbound_services:
- mail
```

开发者设置好这些内容后，其应用程序就可以接收电子邮件了。应用程序会收到发送到your-app-id.appspotmail.com的所有电子邮件。开发者需要为传入的电子邮件设置一个处理

程序。电子邮件会在虚拟网址/_ah/mail/receiver-address上接收。举例来说，如果有人给admin@markcc-chatroom-one.appspotmail.com发送电子邮件，我的应用程序会在网址/_ah/mail/admin@markcc-chatroom-one.appspotmail.com收到该电子邮件。

为了处理该电子邮件，开发者需要为电子邮件网址设立一个处理程序。假设我想给聊天服务添加电子邮件功能，使人们可以给管理地址发送邮件，并且要求邮件自动转发给我，那么我需要做的是添加一个脚本处理收到的电子邮件。因为邮件接收被伪装成HTTP请求，如果想要将其实现为一个简单的RequestHandler，那么就需要将传入的请求解析为电子邮件的信息。但App Engine提供了一个可以继承的InboundMailHandler类，不用手工完成解析。App Engine提供了一个POST的实现，获取传入的消息，并将其解析成一个InboundEmailMessage，然后对该消息调用receive。因此，我需要做的所有工作就是提供一个receive的实现，例如下面的实现：

multichat/chatmail.py

```
import email
from google.appengine.ext import webapp
from google.appengine.ext.webapp.mail_handlers import InboundMailHandler
from google.appengine.ext.webapp.util import run_wsgi_app

class ChatMailHandler(InboundMailHandler):
    def receive(self, mail_message):
        mail.send_mail(sender="admin@markcc-chatroom-one.appspot.com",
                       to="markcc@gmail.com",
                       subject="CHAT ADMIN MAIL: %s" % mail_message.subject,
                       body="Original message from: %s\n%s" %
                       (mail_message.sender,
                        mail_message.body)

chatmail = webapp.WSGIApplication([InboundMailHandler.mapping()])

def main():
    run_wsgi_app(chatmail)

if __ame__== "__ main__":
    main()
```

有了该处理程序后，我通过在app.yaml中添加表项，告诉App Engine它应该将在电子邮件网址收到的消息路由到包含该处理程序的脚本中：

```
- url: /_ah/mail/.+
  script: chatmail.py
```

InboundEmailMessage包含了电子邮件消息的所有头部和主体元素的字段。

- sender：邮件发件人的电子邮件地址。
- to：To头部中的电子邮件地址。
- cc：cc头部中的电子邮件地址。
- reply_to：答复邮件应发送到的电子邮件地址，由Reply-To:头部声明。
- subject：邮件主题。

- ❑ `body`：邮件正文的内容。
- ❑ `attachments`：邮件附件的列表。这是个二元组的列表，其中第一个元素是附件的名称，第二个是附件的内容。

15.5 服务结束语

这一章快速浏览了App Engine中服务的思想。我们已经看到了一组最常见的App Engine服务，包括提供对最常用数据对象的快速访问的服务，实现HTTP请求与其他网络通信的服务，以及一些发送和接收电子邮件和聊天消息的基本服务。在这个过程中，我们知道了服务看起来应该是什么样子的，以及可以如何与服务进行交互。我们看到App Engine如何针对传入的数据实现处理程序，甚至当不是真的HTTP时，使用伪URL和HTTP门面。并且，我们使用一些服务提升了之前构建的应用程序。不但使用Memcache为我们实现的文件系统提供了缓存，而且使用电子邮件服务让聊天应用程序可以接收来自用户的电子邮件。

在下一章中，我们将了解如何使用和构建在App Engine服务器上执行大量计算的App Engine服务。这是我们第一次实现不是完全响应式的应用程序——响应式的应用程序只在用户明确要求时执行操作——我们的应用程序要在服务器上自动运行任务，而不需要用户的干预。在实现该应用程序的过程中，我们将使用更多的App Engine服务：调度任务使其在一定的时间发生，将任务放入队列在App Engine云的某处运行，并将计算的结果发送给其他应用程序。

第 16 章

云中的服务器计算

16

到目前为止，我们的所有App Engine程序都是完全被动的，由请求来驱动。这些应用程序位于App Engine的服务器上，完全不做任何事情，除非用户发起某种请求，才立即响应用户的请求，然后返回，继续不做任何事情。对于很多应用程序而言，这样的行为正是我们想要的。一个聊天室实际上并不需要做任何事情，除了用户请求消息时将消息发给用户。文件系统本身不进行读文件或者写文件，只有在有人特别要求它的时候，它才进行操作。

但是，有时这种被动的方法并不适用。例如，如果开发者正在构建一个电子商务网站，可能想得到每天销售的总结，或者，如果开发者正在构建一个协作式日历应用程序，可能希望能够通知人们即将发生的事件，这些工作就不是被动的。它们需要服务器运行一些不需要请求来触发的代码完成相应的操作，这些代码可能基于时间或者基于某些数据相关的事件。

App Engine为非被动的基于服务器的计算提供了两种机制，一种是按照一个固定的时间表执行操作，还有一种是按照基于事件的JavaScript机制执行。在这一章中，我们将学习这两种机制，并看看如何利用这两种机制在服务器上运行任务。

这是App Engine真正的亮点之一。正如我们已经在本书中看到的，App Engine是围绕HTTP请求处理程序构成的。在服务器计算里也是一致的，开发者还是将其所有的代码编写为简单的HTTP请求处理程序。由于App Engine允许开发者使用这些请求处理程序，所以开发者可以为其需要在服务器端计算的任务构建任意复杂的工作流。这是一种非常优雅的实现方式，在不增加大量复杂性的前提下，开发者能够完成其需要做的任何工作。

我们将从了解最简单的服务器计算方式开始，即按照有规律的时间表执行操作。在此之后，我们将转移到更加强大、灵活的任务队列方面，在任务队列中，通过处理用户请求可以动态完成服务器上的复杂计算序列。

16.1 用 App Engine Cron 调度作业

作为云应用程序的管理者，开发者很有可能想要查明其系统是如何被使用的。他可以从App Engine的仪表板中得到很多这样的信息。但是，开发者常常会发现还需要检查一些应用程序特定的数据。

例如，如果开发者正在运行一个网上商店，可能会想进行每日总结，汇总客户的购买量以及

在这些交易中所获得的利润。App Engine的仪表板不能提供这样的信息。它可以告诉开发者有多少访问者来过其商店，在商店呆了多久，使用了多少CPU资源，修改了多少个数据仓库对象，与其交互耗费了多少带宽，但它并不能显示出消耗了多少库存或者赚了多少钱。

开发者可以通过一个URL查看这些细节，可以向对应的URL发送请求，查看在某一天中的收益。但如果想在经营中每天都看到该报告可怎么办呢？如果不用记住网址、请求报告、然后保存这一系列操作，而让App Engine每天自动生成报告并发送给我们，该有多好啊！

实际上，我们并没有实现网上商店这样的应用，这个例子对于本书来讲有点太复杂。然而，调度任务的过程，如报表生成，对于网上商店与聊天服务来说并没有很大的不同。事实上，这是一个很常见的需要：调度任务的关键就是查看开发者的应用程序在一定的时间段内产生的所有数据，并分析数据。分析的目的不同，对于网店，目标可能是要为店主生成报告；而对于聊天系统，可能就会用于决定是否应该终止一个数天没有任何活动的聊天室。但两者基本目标是相同的，都是要设立一个时间表，在这个时间表所指定的时间分析数据，并采用分析的结果做一些有用的工作。

接下来，我们将使用报表生成作为一个典型的例子，并为聊天服务设置一个报表生成功能，生成一个每天每个聊天室发送了多少个消息的报告。

16.1.1 Cron调度器

我们的应用程序有一个时间表文件。在Java中，该文件被命名为`WEB-INF/cron.xml`；在Python中，是`cron.yaml`。由于我们使用的是聊天应用程序的Java版本，所以将在`WEB-INF/cron.xml`文件中增加一个条目。该文件包含了封装在<cronentries>标签中的条目列表。每个条目是一个<cron>元素。

16

cron条目包含了三个字段，表示为XML子元素。

- <url>
  首先，它有一个URL。正如我们在App Engine中反复看到的，基本上一切内容都封装在HTTP的请求中——几乎所有内容都使用HTTP请求处理程序实现。Cron也不例外。App Engine会根据时间表为该URL生成一个GET请求。
- <description>
  其次，有一个描述。这仅仅是文档性描述，并没有执行时间的效果。描述用来让人们阅读cron文件，以了解任务是什么。
- <schedule>
  关于任务应该何时被调用的文字描述。

时间表的编写看起来像一个英文说明。

- every+数字+单位
  在指定的时间间隔运行。例如，`every 2 hours`（每2小时），或`every 3 minutes`（每3分钟）。
- every时间

在指定的时间运行。例如，every monday 12:00是在每星期一中午运行，every day 00:00是在每天的午夜运行。

- nth 时间 of 月份

  如果开发者想要得到的报告不频繁，则可以以月为单位来指定时间：2nd tuesday（省略月表示每个月），或者3rd Friday of march,june,september,december，即一年运行4次报告。

例如，要使聊天室每天生成一个报告，在cron文件中加入如下内容：

ws2/ReportingChat/war/WEB-INF/cron.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<cronentries>
        <cron>
            <url>/report?to=markcc@gmail.com</url>
            <description>Generate a daily chat usage report.</description>
            <schedule>every day 00:00</schedule>
        </cron>
</cronentries>
```

该报告将通过向/report URL发送GET而生成，并在每天的午夜完成。

16.1.2　实现Cron请求处理程序

请求处理程序的基本代码很简单。我们已经学过如何编写聊天消息的数据仓库查询——这个程序中只要能够进行查询，并返回过去24小时内添加的所有聊天消息即可。然后，我们检查消息的长度，就能得到想要的结果。

严格地说，这种实现是矫枉过正，其实不需要取回所有消息。但是，在大多数分析或者报告的预定任务中，我们希望能够排查最近一段时间中的所有数据，所以这是开发者在这类任务中会使用的基本结构：取回所有内容，然后对其排除，最后生成结果。

ws2/ReportingChat/src/com/pragprog/aebook/chat/server/Reporter.java

```
@SuppressWarnings("serial")
public class Reporter extends HttpServlet {

        Logger logger = Logger.getLogger(Reporter.class.getName());

        @Override
        @SuppressWarnings("unchecked")
        protected void doGet(HttpServletRequest req, HttpServletResponse resp)
                    throws ServletException, IOException {
                String toAddress = req.getParameter("to");
❶               PersistenceManager persister = Persister.getPersistenceManager();
                Query query = persister.newQuery(PChatMessage.class);
                query.setFilter("date >= yesterday");
                query.declareParameters("long yesterday");
                query.setOrdering("date");
```

```
            long yesterday = System.currentTimeMillis() - (24 * 60 * 60 * 1000);
            List<PChatMessage> messages =
                 (List<PChatMessage>)query.execute(yesterday);
            resp.setContentType("text/html");

❷           PrintWriter out = new PrintWriter(new CharArrayWriter());
            out.println("<html>");
            out.println("  <head>");
            out.println("    <title>Chat Usage Report</title>");
            out.println("  </head>");
            out.println("  <body>");
            out.println("  <h1>Chat Usage Report</h1>");
            out.println("  <p> Messages in the last 24 hours: " +
            messages.size());
            out.println("</body></html>");
            out.close();
            String report = out.toString();

            Session mailSession =
❸               Session.getDefaultInstance(new Properties(), null);
            try {
                Message msg = new MimeMessage(mailSession);
                msg.setFrom(new InternetAddress(toAddress));
                msg.addRecipient(Message.RecipientType.TO,
                    new InternetAddress(req.getParameter("to")));
                msg.setSubject("Chat Status Report");
                msg.setText(report);
                Transport.send(msg);
            } catch (AddressException e) {
            // 邮件地址是一有效的常量，因此这种情况不可能发生
            } catch (MessagingException e) {
                    logger.log(Level.INFO, "Error sending report: " + e);
            }
        }
}
```

此代码就是一个简单的servlet——现在，开发者应该不用任何帮助就能够读懂这些代码了。这只是我们以前见过的代码的组合而已：

❶ 首先，我们采用通常的持久性方式组成查询语句，并从数据仓库中取回聊天消息，就像10.1节中的代码一样。

❷ 然后我们利用这些信息生成一个包含报告的字符串。

❸ 最后，使用在15.4节中看到的App Engine的邮件服务，发送该消息。

还有一件事情需要处理。我们告诉调度器在`/reportURL`为应用程序生成一个请求，因而，需要告诉App Engine将该servlet连接到此网址。我们按照往常的方法实现该工作，即通过在`war/WEB-INF/web.xml`的<servlet>元素中添加一个条目，注册该servlet，然后添加一个<servlet-mapping>条目，将该servlet和该URL绑定。

```
<servlet>
    <servlet-name>ChatServletReporter</servlet-name>
     <servlet-class>com.pragprog.aebook.chat.server.Reporter</servlet-class>
```

16

```
<auth-constraint>
      <role-name>admin</role-name>
        </auth-constraint>
</servlet>
...
<servlet-mapping>
    <servlet-name>ChatServletReporter</servlet-name>
  <url-pattern>/report</url-pattern>
</servlet-mapping>
```

16.2　用任务队列动态运行作业

第二种在服务器上执行计算的方法比cron调度更灵活，更有趣。然而，基本元素则采用类似的方式处理：开发者在用户请求之外想要做的每项工作都要设置为在特定虚拟URL上的HTTP请求处理程序。

在使用cron调度器时，我们指定应该根据一个具体的时间表来调用特定请求，但也有很多任务没有必要在时间表里调度。例如，在博客上，会定期检查它有多少次点击以及这些点击来自哪里；如果在数据中得到了任何有趣内容，我就会向邮箱发送一个报告的副本。这种操作没有一个具体的时间表——当看到某些想要参考的内容时，就要求发送一个报告。

App Engine提供的一种服务可以处理类似这样的事情，它被称为*任务队列*（task queue）。任务队列是一种非常通用的机制，可用来做任何最终由用户请求所发起的工作——它并不一定是需要立即完成的工作。例如，如果想建立一个在线日历，则可以让日历应用程序向任务队列添加一些内容以发送电子邮件提醒；任务将被放到一个队列中，队列会指示其应该何时运行。当用户创建日历条目时，该任务就进入了队列，实际的任务直到满足激活条件时才执行。任务队列是个非常强大的机制，那些从任务队列中运行的任务，它们自己可以向队列中添加其他的任务！由此，任务队列就成为一种能够构建任意任务序列和流程的完全通用的工作流系统。

任务队列的基本思路很简单。应用程序可以指定一组命名工作池，工作池包含需要完成的工作，称为任务队列。每个工作单元由一个请求进行描述。App Engine服务器会定期检查任务队列，如果有任何请求在队列中等待，服务器就会触发它们，并执行处理程序。在一般情况下，当开发者把某任务放到任务队列后，App Engine会尽快地运行它，但并不保证要等多长时间。

16.2.1　任务

那么，什么是任务呢？从概念上讲，任务是一个小的工作单元。它基本上是语言中的一个函数，例如在Python中用于接收参数与输入并随后执行计算的一大段代码。从App Engine程序调用一个任务和调用一个函数/方法的主要区别在于，任务是异步的，它与构建Java用户界面所用的GWT方法一样，调用者本质上只需说明“我要完成这个任务”，然后就可以去做其他事情了。调用者不会等待结果。如果调用者需要使用结果做某些事情，那么它必须提供一个回调函数。在常规编程语言如Java中，使用对象描述了回调函数。在任务环境中，回调不是一个对象，而是另一

项任务。如果这听起来有点混乱，不必担心，让我们来看一些具体的例子。

假如正在构建一个协作式日历应用程序。任何用户都可以安排会议，向其他人发送邀请，这些人可以回应、接受或拒绝邀请。当会议发生时，该应用程序向预计与会者发送提醒。

我们在日历上创建一个会议时，可能希望向人们发送电子邮件邀请函，这可以通过为每个邀请函创建一个任务来实现。我们不想在用户等待请求响应时发送邀请函，因为发送邮件的速度很慢！如果有十几个被邀请者，发送十几封邮件会使用户挂起很长一段时间。所以只需创建一堆的任务并将其放入队列即可。这些发送任务会被完成，但不需要等待它们完成。

这就是基本思路。开发者的App Engine代码总是编写为一堆HTTP请求处理程序，或者servlet。如果开发者想在服务器上进行计算，那么就创建某个任务，告诉服务器如何产生请求，这样就完成了工作。编写的任务实现方式并没有什么特别之处，任务只是事件处理程序。事实上，可以创建发送给相同URL的请求事件。任务的实现和任何其他处理程序的实现实际上并没有差异。我们为cron任务编写的报告处理程序作为任务实现，也可以工作得很好。也可以编写提交聊天消息的任务，只需通过在队列中加入一个任务，然后在提交聊天的URL上使用正确的参数执行请求即可，执行方式与用户期望的完全相同。

16.2.2　创建任务

任务如何在代码中呈现？如前所述，它们就是告诉服务器如何创建HTTP请求的简单对象。

因此，如果想要发送电子邮件（正如上一节中，要发送一个聊天室使用情况的日常报告一样），就要定义一个有一个参数的任务，这个参数就是该报告应该被发送到的电子邮件地址。就这么简单。由于报表生成的URL是/report，这就是该请求的URL。我们实现了报告的生成器，以处理GET请求：所以当构建任务时，要将其编写为一个GET。并不需要实现处理程序，因为已经做好了。Cron的工作中生成了一个完全正常的GET，因此，任务可以使用相同的处理程序——它只需要生成相同的、基本的GET即可。

如果希望用户能够在聊天服务中请求生成使用报告，那么可以在用户界面中添加一个按钮，生成一个GWT远程过程调用。该调用调用了generateReport方法：

```
@Override
public void generateReport(String address) {
❶        TaskOptions task = method(Method.GET).param("to", address);
❷        QueueFactory.getDefaultQueue().add(task);
}
```

❶ 此代码有一些奇怪的地方。在App Engine中，我们通过创建描述任务的TaskOptions对象来提交任务。但是，我们并没有显式地创建该对象。相反，我们调用了类com.google.appengine.api.labs.taskqueue.TaskOptions.Builder中的静态方法。为创建一个GET任务，我们调用了包com.google.appengine.api.labs.taskqueue.TaskOptions.Builder中名为method的方法。但这种实现方法需要输入很多字符。在一般情况下，在任务队列的代码中，我们只需做一个静态导入，将

```
import static com.google.appengine.api.labs.taskqueue.TaskOptions.Builder.*;
```

放在我们的代码导入部分，该静态方法就变得可用了。只需要使用`method(Method.GET)`就可以创建一个GET任务。该方法将创建`TaskOptions`对象，然后再使用选项对象的方法设置它的参数。

❷ 提交任务执行也很简单，只需使用`QueueFactory`的方法获取队列——`getDefaultQueue()`用于获取缺省队列，或者`getQueue(name)`用于获取其他任务队列。然后，向任务队列添加任务选项对象。

我们可以配置以下任务选项。

- **任务的名称**

  当我们为任务队列创建一个任务时，可以给它分配一个名称。该名称将出现在应用程序的Google App Engine控制面板的错误、日志和任务监视器中。
- **请求的类型**

  HTTP请求像往常一样，可以是GET、PUT或POST请求。我们使用`method()`方法设置HTTP请求的类型。
- **请求的URL**

  我们不用直接声明任务应该运行的代码，只需声明URL。App Engine发送相应的请求到该URL，调用为该URL设置的请求处理程序。我们使用`url()`方法设置请求的URL。
- **CGI参数**

  CGI参数是会被编码到请求的URL中的参数。在一般情况下，使用简化的CGI参数，不包括标点和空格的简单值。在我们的例子中，事件标识符使用CGI参数，事件标识符应该是简单的值，如整数。如果需要，也可以声明许多CGI参数，对于每个CGI参数，调用`param()`方法将其添加到任务中。
- **头部参数**

  这些是在HTTP请求的头部行中声明的参数。头部参数是单行字符串参数，可以有空格、标点符号，以及其他任何我们想要的内容。和CGI参数一样，如果需要，也可以声明很多头部。通过调用`header()`方法将头部参数添加到任务中。
- **消息体**

  因为我们正在处理HTTP请求，所以，如果请求是PUT或者POST，就会有一个消息体，从而，可以在消息体中放入我们想要放的任何内容。可以使用`payload()`方法设置消息体。
- **估计的执行时间**

  通过设置任务的`eta`（预计抵达时间），可以告诉App Engine，在一个特定的时间之前，我们不想执行该任务。App Engine的服务器会将该任务保留在队列中等待，直到`eta`到了之后。`eta`之后，服务器会尽快执行该任务。

从处理程序方面来看，任务更容易——本节中反复说过，它们只是标准的事件处理程序。但有时开发者可能需要判断一个给定的请求是由人还是App Engine事件产生的。例如，如果聊天应用程序中有一个反馈框，当用户提交反馈时，应用程序将使用与上边展示的相同处理程序来处理

反馈。当有人按提交按钮时，他会期望某种反馈，比如一个消息，说“你的反馈已经邮寄给系统管理员”。但是，如果该事件是作为一个服务器计算任务被调用的，应答中所包含的任何内容都会被丢弃，那么何必要麻烦地为服务器生成应答呢？

开发者可以通过查看消息头部来判断任务是何时生成请求的。每一个任务队列生成的请求有3个头部，描述了调用它的任务。

- X-App Engine-QueueName 创建任务的队列的名字。下一节中将看到，我们可以有多个任务队列，每个队列都有其自身的配置。
- X-App Engine-TaskName 用来标识该任务的名称。
- X-Appengine-TaskRetryCount 当任务队列调用一个任务时，会等待10分钟完成任务。如果在这个时间内没有完成，或者，如果请求返回一个错误码，系统将重试。这个字段通常为0，但是，如果任务之前失败了，而当前执行的是一个重试，这个字段则表明任务失败了多少次。（超时时间在写本书的过程中有着相当显著的变化，从30秒到当前的10分钟不等，因此，如果超时时间对你真的很重要，你应该检查目前Google App Engine的任务队列文档，核对当前的超时时间是多少。）

16.2.3 使用多任务队列

正如上节所述，可以有多个任务队列。不过，既然单个队列就可以处理任意多个任务，而且每个任务也可以声明自己的目标URL和参数，所以读者可能会疑惑为什么需要多个队列。

多数时候可能并不需要多任务队列。对于许多应用而言，使用单一任务队列就足以完成所有的服务器计算。但也有一些工作是只能以队列方式配置，而不是以单个任务方式实现的。请记住，开发者必须要为执行任务付费。如果开发者有一个相对昂贵任务（例如，发送电子邮件），可能希望限制队列执行的任务数。可以配置一个队列，例如使其每秒只处理一个事件。由此，通过限制其系统可能发送的邮件数量，最终就能控制为邮件服务所支付的费用。

可能需要使用多个队列的另一种情况就是处理并发。在一般情况下，App Engine从队列中按顺序运行任务，也就是说，插入到队列的第一项任务是第一个被执行的任务。如果队列中有一百个事件，然后再添加另一个任务，那么，这个任务在它之前的一百个任务执行完之前不会被执行。如果有一个高优先级的任务需要马上执行，就完全无法用一个队列完成，相反，需要一个独立的、高优先级的队列。普通的任务发送到默认的队列，而发送到较高优先级队列的任务不等默认队列中的任务执行完就可以先执行。

每个App Engine应用程序都有一个默认的队列。如果只是想使用默认的队列，那么不需要做什么特别的工作。但是如果想要使用多个队列，那么通过给应用程序添加一个queue.xml文件就可以创建其他队列。例如，可以使用queue.xml生成一个默认队列和一个高优先级队列：

ws2/ReportingChat/war/WEB-INF/queue.xml

```
<queue-entries>
    <queue>
```

```
        <name>default</name>
        <rate>5/s</rate>
    </queue>
    <queue>
        <name>priority-queue</name>
        <rate>10/s</rate>
    </queue>
</queue-entries>
```

每个队列具有以下可以定义的特性。

- **名称**

  任务队列的名称。

- **速率**

  该队列处理任务的最大速率。使用每秒多少个进行声明（10/s表示每秒10个，5/m表示每分钟5个，1000/d表示每天1000个）。

- **总存储限制**

  在队列中等待的任务可使用的总存储空间。例如，10m表示10兆字节。这不是任务的处理程序可以使用的空间，那是由App Engine的设置决定的。作为任务描述的一部分，该设置仅仅是指可以放在队列中的信息数量。

16.3 服务器计算结束语

在这一章中，我们讨论了在用户请求之外，如何在App Engine的服务器上运行任务。事实上，它很简单，开发者只需要提供一组HTTP请求处理程序，或者servlet，就像为其他HTTP请求所提供的处理程序一样，然后，开发者可以简单地在任务队列中加入一个条目，告诉服务器如何生成请求，并运行新的任务。

我们没有讨论的一件事情是如何管理谁可以在哪个URL上发送请求。使用任务队列，开发者已经获得了所需的URL，提供给服务器计算任务一个切入点。开发者不希望用户能够向这些URL发送请求，无论是说会提供给人们一个机制使其能够访问他们实际不应该访问的数据，还是说会引发拒绝服务式攻击，这都是一个重大的安全漏洞。这个问题在App Engine中尤其严重，因为开发者要为其所使用的资源付费。如果有人能够使用未经授权的访问进行攻击，使开发者的应用程序使用了大量的CPU时间，那么，开发者会面对一个非常巨额的账单。

下一章将详细地看看安全问题。我们将了解安全的真正含义是什么，以及如何使用App Engine构建一个安全的系统。

第 17 章

App Engine服务的安全性

本章来学习安全性，这是一些我们还没有真正涉及过的内容。安全性是一个很大的话题，要完全覆盖这个话题，其篇幅会超出整本书。但是，我们可以快速了解一下相关的基础知识：安全性的含义是什么，在App Engine内如何实现基本的安全设施。

17.1 什么是安全性

安全性的基本概念很简单，因为每一个应用程序都使用某一组数据对象进行工作，安全性作为一种策略，建立了一套规则，定义了谁可以查看或修改这些数据对象，以及以何种方式查看或修改这些数据对象。在一个安全的系统中，任何数据对象都不能被未经安全策略专门授权的用户查看或修改。

例如，在聊天室应用程序中，允许任何人访问他们想要访问的任意聊天室，但只能在登录之后才能访问。所以我们已经有了一个非常薄弱的安全策略。聊天室中有三种操作：查看聊天列表、阅读聊天信息，以及发布聊天消息。只要登录进去，任何用户都能执行这些操作。

对于许多应用程序而言，这种安全性策略还远远不够。例如，想象一个购物应用程序。用户应该只允许查看和编辑他们自己的购物车。任何人都应该能够查看目录，但是，应该只有店内员工能够编辑目录。大多数员工应该能查看购物车，以帮助购物者，但他们不应该能够改变购物车的内容。此外，应该只有老板或经理能查看整体业务的财务状况。

但正如读者从这个购物应用程序的例子中所看到的，我们仍然可以从对象管理的数据集合以及对数据进行的操作这两项角度来施加必要的安全策略。

17.2 基本的安全性

在最基本的层面，安全性由两部分组成，一部分是定义安全策略，另一部分是使用登录认证和检查去强制执行这一策略。安全策略定义了究竟允许谁做什么，通过登录认证来识别用户，授予用户应享有的访问权，检查则可以拒绝用户访问无权限内容。

本节将通过一个例子来介绍如何定义基本安全策略，以及如何使用App Engine的设施实现这一策略。我们将给聊天应用程序添加一个管理设施，并定义安全策略，对使用该应用程序的用户加以控制。

17.2.1　添加聊天室的管理功能

下面还是通过聊天应用程序来简单地介绍基本安全性。在我们原来的聊天应用程序中，可用的聊天室集合是固定的。当应用程序首次运行时，会自动初始化一个聊天室的小列表，这是所有被允许使用的聊天室。为了改变该列表，我们需要修改聊天应用程序的源代码。

与以前不同的是，我们希望能够动态创建新聊天室，但不希望任何用户都能够随意乱糟糟地创建聊天室，要对用户的权限加以控制。

先来定义一个安全策略，描述一下应用程序所要用的基本资源及操作的集合。然后再来学习一下如何在App Engine中实现这些策略。

安全策略有三种对象：聊天室、聊天消息和用户。只围绕其中的两个来定义安全策略：用户和聊天室。单个的聊天消息由它们所属的聊天室进行控制。实际的安全性将通过控制聊天室来提供。

我们在这里要小心。开发者常犯的一个最大的失误是：创建了一个安全策略，它定义在一个高层次的复合对象上（就像我们的聊天室），但后来却提供了一些通过另一种接口访问对象的方式，而这些接口并不是以安全策略定义的对象定义的。这样的备用接口由于不能提供可用的安全机制（可能是无意的，或者是蓄意攻击的一部分）而违反了安全策略。

例如，在聊天界面中，我们勾勒出一个基于时间的聊天视图。我们可以将其扩展为一个完整的基于时间的接口，构建显示最后30分钟内发布的所有信息的视图。假设实现了该视图，我们提供了一个只是按时间戳取回所有消息的视图。如果我们有一个基于聊天室的安全策略，只在用户要访问聊天室时检查用户的权限，那么，该时间视图可能完全使这一安全策略无效，用户可以通过查看时间视图查看到任意聊天室的内容——即使按照安全策略他们没有权限查看。

这就是使安全性变得棘手的地方，开发者需要考虑周全。这些微小琐碎的错误很容易出现在系统中的一些不起眼角落，然后，一切轰然倒塌，所有的安全策略都没有意义了，系统变得彻底不安全。

现实世界中就有大量这样的例子。说真的，大多数软件病毒利用的正是这种漏洞。软件中的安全系统用来防止程序做某些类型的操作，但在系统的一些小角落中却并没有考虑如何与安全策略交互，以至于出现了各种漏洞。

正如开发者所看到的，确保仔细全面地设计安全策略非常关键。事实上，读者在本书中看到的方法并不是在实际应用程序中所遵循的过程。我们要做的是把安全性加到现有应用程序中，而在实际应用中，开发者必须从一开始就考虑安全性，安全策略应该是设计的首要内容之一。

对于这个聊天应用程序，我们将竭尽所能设计一种策略，就好像是为一个新应用程序设计那样。我们将采用Java版本的应用程序，因为这是一个比较简单的聊天应用程序的实现。只需要考虑应用程序中有哪些视图，然后设计一个安全策略，涵盖应用程序中所有可能的用户访问数据的方式，并且要绝对确保每个视图的实现都考虑和执行了该安全策略。尽力保持简洁，创建如下三个角色：

- 普通用户，可以在聊天室读取和发布消息；
- 特权用户，可以做普通用户能做的所有事情，还可以创建新的聊天室；

- ❑ 管理员，可以做特权用户能做的所有事情，还可以创建新的特权用户。

缓冲区溢出

有一个已被广泛利用的安全缺口，那就是缓冲区溢出。虽然在 App Engine 中开发者不必担心这种错误，但作为一个案例研究，缓冲区溢出还是很有趣的。

在大量的 C 代码中，数据会被读入缓冲区，即一块预先分配好的内存。例如，如果要做一个输入框，那么，很多 C 语言框架会让开发者传递一个指向缓冲区的指针，而输入框中的内容则将被写入该缓冲区。

缓冲区的大小是固定的，因为它在传递之前就被分配好了。比如，开发者有一个输入框，它可容纳 80 个字符的字符串，那么就可以传递一个 80 个字节的内存缓冲区。

但如果有人返回 81 个字符而不是 80 个，又会发生什么呢？或者，如果返回了 800 个字符呢？

这正是缓冲区溢出发生的情况。攻击者将多于已有缓冲区空间的数据填入到缓冲区。如果开发者的代码不检查长度，只是简单地复制数据，那么开发者最终会用复制的数据覆盖其程序的内存，从而改变了数据结构，甚至可能在内存中插入新的代码，而这些插入的代码会被执行！

就我们一直所讨论的内容来看，这怎么算是一个安全漏洞呢？这其中涉及了安全策略的重要方面。在安全策略的较底层——基础的编程语言和库所提供的安全性的级别——安全策略应该是没有人能向不属于其的内存中写东西。但是，草率的代码实现方法并没有验证是否遵循了该策略。该策略要求每个向内存中的写操作都要检查该写操作是被允许的，但代码中的实际检查则是：“这个缓冲区的起始地址是否对该调用来讲是一个有效可用位置？”安全策略被打破了，因为检查只查看了实际策略条件的一半。正确的检查应该是：“我可以复制整个字符串数据到这个缓冲区吗？”该检查由两部分组成：“这是一个有效的缓冲区地址吗？我复制的数据能刚好放入这个缓冲区吗？”

17.2.2 实现聊天角色

App Engine有一个与应用程序相关联的角色概念。默认情况下，每个应用程序都有两个角色：管理者，以及正常登录的用户。对于这些内置的角色，登录认证的管理非常容易。在应用程序的配置文件中，开发者可以标记某些网址为只允许管理员使用。在5.2节中，我们看到了如何使用Users服务进行配置，实现一个页面要求通过认证的、登录的用户才能访问，即通过在`app.yaml`文件中加入`login:admin`代替`login:require`。

对于更多类型的角色，开发者需要自己建立。基本上，需要创建一个包含角色数据的持久性对象，然后由请求处理程序取回角色数据，并在生成页面之前对其权限进行验证。这就是聊天应用程序下面所应做的事情。

任何可以登录到Google的人基本上都是普通用户。因此，针对普通用户页面，我们需要实现的就是已经设置好的要求用户登录。针对特权用户和管理员则还需要做其他的工作。特权用户和管理员都能访问普通用户所不能施行的操作。因此将为这个应用程序再添加两个新页面，以表示这两个新视图：一个允许特权用户添加新的聊天室，一个用于管理员管理用户。我不打算仔细描述这些页面的完整的用户界面实现，它们的形式和之前实现用户界面相同，我们只用占位符表示这些页面。这里最重要的是了解如何保护这些网页，以确保它们只能以安全策略允许的方式进行访问。

根据我们的安全策略，任何登录的用户都可以访问应用程序中的普通聊天视图页面。所以策略的这一部分已经实现，我们已经使用App Engine的Users服务要求用户登录，没有人能够不登录就访问视图。

为了能够实现安全策略，需要有一个机制能够识别哪些用户具有除默认的登录用户角色以外的角色。要想实现该功能，需要给数据仓库再添加了一个新类型UserRole。该角色类型仅仅是一个用户名（登录用户的名字，作为登录服务的返回）和一个包含用户角色的字符串。该类型以及取回用户角色的函数的基本实现如下所示：

secure-chat/tchat.py

```
class UserRole(db.Model):
    name = db.StringProperty(required=True)
    role = db.StringProperty(choices=["User", "admin", "privileged"],
                             default="User")

    @staticmethod
    def GetUserRole(name):
       user_record = db.GqlQuery("SELECT * from UserRole WHERE " +
                                 "name = :1",
                                 name)).get()
       if user_record != None:
         return user.role
       else:
            return "User"
```

这些代码非常简单，该类是一个有两个字符串字段的标准的数据仓库类型。通过用一个包含可能值的特定列表注释role字段，只有列表中的那些角色允许访问。任何人都不可能通过一个错误的，或者一个恶意的攻击，用一个无效的角色创建用户。而且，任何不在数据仓库UserRole记录中的用户都无法通过检查，因此会以默认的角色User结束。

这种默认的规则由GetUserRole执行，它为一个特定的用户返回角色。如果用户没有被授予一个特定的角色，那么他们将在UserRole表中不会有表项，因此，GQL查询最终将引发一个IndexError。当发生IndexError时，GetUserRole最终会运行异常处理代码，返回默认的角色User。

这段代码中一个有趣的事情是：角色特性中有限值的使用。我们并没有让role角色使用无限制的文本字符串特性，相反，我们特意对其进行了限制，确保用户有一个有效的角色。GetUserRole不可能返回应用程序所识别的三种有效角色之外的内容。这一点看起来似乎并不重

要，但是当开发者实现安全性时，这绝对至关重要，开发者对所有细节都得超级偏执。如果只有三种可能的角色，那么开发者可以绝对确保其代码中出现的只有三种可能的角色值——即便如此，开发者仍然得进行检查。

接下来，一个角色受限的页面被请求时，我们需要检查用户是否具有针对该页面的正确角色：

secure-chat/tchat.py

```
    def ValidateUserRole(actual, required):
❶       if required == "admin":
          return actual == "admin"
❷       elif required == "privileged":
          return (actual == "admin" || actual == "privileged")
        elif required == "User":
          return True
❸       else:
        return False
```

同样，这段代码非常简单。但是，就像通常的安全性代码一样，它的编写显得非常死板。对于管理者页面，我们只在用户是admin时显示该网页；对于特权页面，我们只在用户是privileged或者admin时显示该页面。即使是第三种角色，User，虽然实际上并不需要检查，我们也把它放到其中。为什么要如此死板呢？设想以后增加了一个超级管理者角色，而忘记更新这个功能。如果我们只是顺着代码执行下来，没有检测User就返回True，那么需要超级管理者角色才能访问的页面的验证请求最终会进入与User相同的缺省情况，即允许任何人访问它！ 我们特意检查了每个角色，以保证没人通过注入非标准的角色来试图欺骗应用程序。

对于应用程序的其余部分，需要考虑用什么类型的用户界面元素提供安全策略所需的能力。安全策略表明特权用户允许创建新的聊天室，而我们的应用程序目前还没有这样的方法，所以需要创建一个新的页面，提供创建新聊天室的功能。该功能的实现代码参见code/secure-chat/new-chat.html。

基本的请求处理程序是我们惯用的典型的App Engine代码。除了在开始时的角色检查，它就是一个标准的处理程序。除此之外，它只是一个能够生成标准表格的处理程序，实现POST请求的提交。

不同的是，在向用户显示页面之前，需要检查用户有必要的权限。我们采用如下所示的代码实现此功能：

secure-chat/tchat.py

```
    class NewChatRoomHandler(webapp.RequestHandler):
❶     @login_required
      def get(self):
        user = users.get_current_user()
❷       role = GetUserRole(user)
        if not ValidateRole(role, "privileged"):
            self.response.headers["Context-Type"] = "text/html"
```

```
            self.response.out.write(
                "<html><head>\n" +
                "<title>Insufficient Privileges</title>\n" +
                "</head>\n" +
                "<body><h1>Insufficient Privileges</h1>\n" +
                "<p> I'm sorry but you aren't allowed to " +
                "access this page</p>\n" +
                "</body></html>\n")
        else:
          self.response.headers["Content-Type"] = "text/html"
          template_values = {'title': "MarkCC's AppEngine Chat Room",
              }
          path = os.path.join(os.path.dirname(__file__), 'new-chat.html')
          page = template.render(path, template_values)
          self.response.out.write(page)
```

角色检查如❷所示。由于我们使用了几个基础设施的方法来构建角色代码，所以，检查角色并不难。只要调用GetUserRole取回用户的角色，然后调用ValidateRole验证该用户是否具有访问该网页的权限即可。我们还使用了一个非常好的快捷方式，没有去显式地检查用户是否确实已经登录了，而是使用了❶处的装饰器（decorator）。装饰器是Python的一个标准功能（Java也有，在Java中称为注解），它允许开发者给代码附加元数据。在这个例子中，该装饰器（由App Engine提供）告诉网络应用框架，该方法应该只有在用户已经登录时才能调用，如果用户没有登录，自动将用户重定向到登录页面。

简单来看，这些功能可能看起来已经足够了。但事实上，要建立一个正确的系统，这还不够。从用户界面方面来看，如果用户没有权限，他就不能进入管理页面，而且如果没有必要的角色，用户也就不能创建新的聊天室，所以这应该是安全的。

但想要破坏系统的人并不一定要遵守其用户界面使用方式。如果要进行修改，应用程序会向服务器提交一个POST请求。狡猾的攻击者可能会注意到特权用户，并看到用于POST管理者的特权级修改的URL，然后，该攻击者直接向此URL提交一个POST请求。因此，该页面的GET处理程序和更新的POST处理程序都需要通过验证过程来保护，以保证只有适当的角色的用户才能提交相应的请求。

所以，POST处理程序与GET处理程序需要以完全相同的方式检查权限。

secufre-chat/tchat.py

```
class NewChatRoomPostHandler(webapp.RequestHandler):
  @login_required
  def post(self):
      user = users.get_current_user()
      role = GetUserRole(user)
      if not ValidateRole(role, "privileged"):
          self.response.headers["Context-Type"] = "text/html"
          self.response.out.write(
              "<html><head><title>Insufficient Privileges</title></head>\n" +
              "<body><h1>Insufficient Privileges</h1>\n" +
              "<p> I'm sorry but you aren't allowed to access this page</p>\n" +
              "</body></html>\n")
```

```
else:
    newchat = cgi.escape(self.request.get("newchat"))
    CreateChat(user, newchat)
    self.response.out.write(
        "<html><head><title>Chat Room Created</title></head>\n" +
        "<body><h1>Chat Room Created</h1>\n" +
        "<p> New chat room %s created.</p>\n"
        "</body></html>\n" % newchat)
```

与创建聊天室的方式类似，我们将使用有权限检查的表单去创建新的特权用户。要成为一个特权用户，必须由管理员授予该用户特权角色。所以就像在新聊天室中一样，我们创建了一个GET处理程序，生成表单；还创建一个POST处理程序，该程序实际上完成了创建特权用户的工作。

综上所述，这就是安全性的基本道理：定义一个策略，然后建立执行该策略的系统。基本思想确实不复杂，开发者在系统中定义哪些对象可以访问，谁可以查看/修改它们，以及以何种方式查看/修改。这就是开发者的安全策略。然后，开发者要进行检查，以确保这一策略得到遵守。可是，现实情况是，实现一个安全系统极其困难。开发者需要知道所提供给用户的每一个操作的影响，无论是直接的还是间接的，并且，开发者需要确保其代码实现在任何情况下都能严格地遵守所制定的安全策略。

到目前为止，我们谈论的主要内容是一个协作式系统的简单安全性，即假设该系统的用户是协作的，而不是主动试图破坏或者禁用系统的。实现这样的系统已经很难了，但还没有完全反应现实，因为除此之外，还有人会试图以破坏开发者的系统为乐。因此，开发者需要考虑一些更复杂的情形：保护自己免受攻击。在本章的其余部分，我们将了解一个恶意用户可能用哪些基本攻击方法试图禁用系统，以及开发者可以用什么机制来保护其系统免受这些攻击。

17.3 高级安全性

安全性的更复杂的一个方面是防御*攻击*（attack）。攻击是有人未经安全策略允许就尝试访问数据或者进行更改，以及阻碍他人按照安全策略允许的方式行事。

攻击可能非常狡猾且富有想象力。我们没有办法完整讨论开发者的服务可能受到的所有攻击方式，只是概述一些典型例子，涉及攻击可能使用的基本通用的技术。

从广义上讲，可以按照攻击试图要做什么操作进行分类。

- **直接攻击**（Direct Attack）
  这种攻击会欺骗系统去实施原本不允许做的事情。基本上，这类攻击并非试图基于网络基础设施进行某种精细的欺骗，而是找到开发者的安全策略的缺陷，然后利用这些缺陷实现攻击。我们将在17.3.1节中更详细地了解这种攻击。
- **跨站点脚本攻击**（Cross-site Scripting Attack）
  跨站点脚本攻击（XSS）非常常见，也十分危险。在XSS中，攻击者在其请求中包括一些HTML页面和JavaScript脚本。如果开发者允许用户提交会直接出现在应答页面中的内容，而没有做任何验证，那么攻击者就能够欺骗应用程序，实现他们想做的几乎任何事情。
  这个聊天应用程序非常容易受到这种攻击：因为不检查聊天消息的内容，攻击者就可以

在聊天消息中插入HTML脚本标记，该消息可能导致攻击者欺骗其他用户的浏览器执行请求。从而，攻击者可以伪装成任何其他用户！我们将在后续页面中的17.3.2节中，了解如何保护自己免受XSS攻击。

- 窃听攻击（Eavesdropping Attacks）

  这种攻击（包括了其常见的变种“中间人”攻击）试图使其位于通过身份验证的用户和系统之间，查看通过身份验证的用户有权访问而攻击者无权访问的数据。这种攻击也可以用来窃取证书，也就是说，找到攻击者可以利用的信息，从而把自己伪装成一个有效的用户。这种攻击要依靠开发者自己来防护是非常困难的，因此，加密协议就是设计来帮助解决这个问题的。后续章节将介绍如何使用SSL（安全套接字层）实现窃听防御。SSL并不是一种完整的防御，但作为一种精心设计且比较周密的安全策略中的一个组成部分，它确实是一个比较好的初始方案。

- 拒绝服务攻击（DoS，Denial-of-Service Attack）

  与我们已经讨论的其他攻击相比，在攻击方式上，拒绝服务攻击并不算是真正的攻击。它不会尝试访问或者擅自改变数据，而是试图阻止授权用户访问他们应该有权访问的数据。拒绝服务攻击最常见的形式是让成千上万感染了病毒的计算机，向某个应用程序发送庞大的伪造请求。应用程序不得不花费大量的时间来处理这些伪造请求，使得有效的用户无法实际访问任何内容。

  要特别小心防范对开发者的App Engine代码的DoS攻击，因为开发者要为用来处理请求的CPU时间付费。开发者花在一个无效的请求上的每毫秒的时间都可能是昂贵的，一个请求耗费的额外1毫秒CPU时间可能在典型的DoS攻击中很容易就累计成额外的1小时CPU时间。幸运的是，App Engine的服务器管理者会监测流量模式，从而能够发现DoS攻击，而且通常在攻击到达开发者的应用程序之前就会关闭该攻击。但是，开发者还是应该有所警惕，总是尽早地检查所收到的请求，进行各类最基本的验证（在开发者做任何其他事情之前进行），并且需要拒绝无效的请求，不浪费任何资源。我们将在17.3.4节中，简要地了解一些由App Engine提供的工具，帮助开发者处理DoS攻击。

17.3.1 直接攻击

直接攻击是最容易防御的攻击。直接攻击直截了当地企图欺骗开发者的系统，让用户执行他们无权执行的某些操作。例如，在上面的例子中，攻击者可能尝试未经批准就直接通过发送`POST`命令给服务器，使用创建聊天室的对话框。

对于这种攻击类型，我们的防护方式是提供精心定义的安全性策略，并仔细实现该策略。这是一种最容易防御的攻击——虽然如此，实现起来也可能非常困难。正如上一节中所讨论的那样，我们需要确保每一个处理程序总是检查用户的身份验证，检查用户是否有权限执行特定的操作。另外，最重要的是，保证被请求的操作是有效的。大多数系统中最常见、最成功的攻击类型就是使用无效请求的直接攻击。如果想要使其系统变得安全，就要无一遗漏地检查所有的请求。允许无效请求就意味着提供了一个漏洞，而一个足够聪明的攻击者能够找到这个漏洞，并将其尽量放

大。例如，在聊天室例子中，必须检查发送消息的用户名称，确保用户确实已经登录，确认他们试图发布消息的聊天室确实存在，并确认他们的消息被正确地格式化为XML请求。如果这些内容中有任何一点不符合，那么应用程序都不应该继续处理该请求。

17.3.2 跨站点脚本

跨站点脚本是一种利用云应用程序开发者的草率之处而进行的攻击。由于HTML页面由简单文本和易于输入的标记所组成，所以请求的主体可能会包含HTML标记。然后，使用主体包含的HTML标记，可以提供JavaScript代码。

假如用户发送包含如下内容的聊天消息，会发生什么呢？

```
Hi there <script language="javascript"> print("JavaScript!");</script>
```

用户的显示屏上会显示聊天消息“Hi there JavaScript！”——但是打印单词“JavaScript”的JavaScript代码会在用户计算机上执行。

还有一种密切相关的跨站点脚本攻击，试图在服务器上运行代码。这种攻击最常见的形式称为SQL注入。具体来说，SQL注入并不是App Engine的问题（因为开发者并不使用SQL），但它确实是一个非常典型的一般性攻击。

在SQL注入攻击中，攻击者向查询语句中作为字段的表单发送了一个字符串。例如，在许多系统中，用户在登录时，会被要求在表单中输入用户名和密码，然后系统将执行一个SQL查询取回正确的密码。系统会做一个查询，如：`SELECT password FROM Users WHERE name =$1`，然后，用户可以提供一个用户名，如`me;DROP TABLE *.`。如果用户的用户名字段值在查询语句中被直接使用，会执行为：`SELECT password FROM User WHERE name=me; DROP TABLE *.`。攻击者知道该字符串将直接被传递给SQL查询语句，所以，可以创建一个字符串执行其他具有破坏性的SQL语句。

防御这两种攻击的方式很明确，就是总是清空用户的输入。也就是说，当开发者从一个用户那里得到输入时，总是获取该输入，并进行清空过程，消除所有元字符。为了防止有人在开发者的HTML中注入JavaScript，开发者应该确保所有<字符都被XML文本字符替换，如`&lt`。为防止有人注入SQL，开发者需要确保其输入不包含类似引用、查询元字符或语句结束符；如果有，这些字符就需要转义。

App Engine提供了可以用来防御XSS类型攻击的工具。如果开发者使用Python，网络应用框架的`cgi`模块中有一个函数，只要调用`cgi.escape(input_string)`，开发者就不必担心这样的XSS攻击了。在开发者的模板中，开发者也可以在输出端添加一个过滤器进行转义。如果给Django模板中的任意变量追加了`|escape`，该变量的值就会针对HTML进行正确转义，防止XSS攻击。对于Java，则稍微复杂一些。如果开发者正在使用GWT进行所有工作，就已经被保护了。GWT处理了XSS的所有情况，并摆脱了使用RPC做的所有事情中的问题。如果开发者不使用GWT，那么处理XSS的方式取决于开发者使用的库，但是，几乎每个框架都提供了一些内置的方法。例如，如果开发者使用Java Server Pages（这是和我们在第6章中使用的Django模板系统的Java版本相类

似的类型），那么有一个标准函数`c:out`用于打印值，只要开发者给`c:out`调用添加一个属性`escapeXml="true"`，它就会自动负责处理好字符串。

17.3.3　窃听攻击

窃听攻击非常微妙，也非常难对付。这种攻击不是试图对开发者的系统做任何事情，而是隐藏在后台并进行观察。为防御窃听攻击，开发者可以做的主要工作就是设计一个精细的、彻底的安全策略，并在所有特权通信中都使用加密通道。

有几种提供加密通道的方法。最简单的就是让App Engine为开发者处理加密，即告诉App Engine对某个特定的URL的所有请求强制使用SSL（安全套接字层，一种标准的互联网加密系统）。要告诉App Engine为一个特定的URL使用SSL，只需在该URL的`app.yaml`条目中添加`secure: always`行。

谈到加密，我可以给读者一个绝对重要的忠告：不要自己实现。如果只要在`app.yaml`中使用`secure`条目就可以实现加密，那么就要这样做。如果由于某种原因，开发者需要更复杂的功能，那么，可以找一种广泛使用的、支持较好的提供加密服务的库。Python和Java中都有扩展的加密库，开发者可以使用它们在App Engine中做加密工作。所以，就使用这些加密库吧！要实现正确的加密非常棘手。如果开发者实现了自己的加密系统，那么，几乎可以确定其中肯定有错误。作为一个加密实现程序，开发者必须绝对保证一切完全正确，然而这其中有许多微妙之处，开发者想要自己做到一切完全正确，几乎是不可能的。即使由许多位专家一起建立的加密系统，通常也会有错误。当有人试图建立一个安全的加密系统时，需要找出其计划中的每一个可能的漏洞，并补上这些漏洞，但是，作为一个试图打破加密系统的攻击者，他只需要找到一个漏洞即可。攻击者的工作远比开发者的工作更容易！因此，请使用其他人已经编写好的通过测试的、认真分析过的，并通过真实考验进行了压力测试的系统。

17.3.4　拒绝服务攻击

正如我之前解释过的，拒绝服务攻击是使用大量请求来堵塞开发者网站的攻击方式。请求可能是完全合法的，也可能是无效的。无论其是否合法，攻击目的都是迫使网络应用程序花费大量时间处理洪水一般的请求，从而导致合法用户无法使用应用。

这种攻击不仅可以通过负载使网站无法访问，而且还会浪费开发者的资金。在App Engine中，开发者要为其使用的资源付费，直到用完所有配额。一旦超出配额，就不会产生更多的账单，但应用程序也将被完全关闭。所以，确实要防御这样的攻击。

防御DoS攻击的一个要点就是尽快验证请求，并拒绝无效请求。如果开发者每秒被10 000个请求击中，那么这种方法不会有多大用处，但对于较小的攻击，则情况明显会得到改善。当开发者收到一个请求时，应该做的第一件事情就是检查该请求是否是有效的。（如果开发者使用GWT，将由框架为开发者处理好检查工作，GWT的RPC会进行请求的自动验证。）开发者应该确保尽量尽快地完成这项工作：如果每一个请求应该有两个头字段，那么在开发者花时间验证这些

字段的内容之前，首先检查是否有这两个字段。如果遗漏了一个或追加了一个额外的字段，就可以拒绝请求，而无需花费更多的时间。

如果开发者面对的是一个真正的DoS攻击，那么通常会看到来自极少数IP地址的消息洪流。这些IP表示的通常是一组已经被攻击者征用的，并被迫向开发者发送请求的计算机。App Engine提供了非常方便的阻塞消息洪流的方式。在应用程序配置中，开发者可以声明一个黑名单（blacklist），就是很多IP地址或IP地址范围的列表，这些地址会被App Engine服务器阻塞。使用DoS黑名单，App Engine会在请求到达开发者的应用程序之前就阻塞它们，从而保护开发者不在攻击者的请求上浪费宝贵的资源。

App Engine的DoS服务非常新，很可能会迅速发展。在当前App Engine版本中，开发者通过在其应用程序配置中加入一个文件进行DoS服务配置，该文件声明了黑名单的内容。在Python应用程序中，开发者创建一个文件名为`dos.yaml`的文件，该文件包含黑名单条目列表。不幸的是，这些条目当前完全不可读，它们是基于称为CIDR（无类域间路由，classless inter-domain routing）的标准的IP路由符号实现的。开发者需要查找CIDR符号才能够编写这个文件。例如，要表明“所有以192.168.0开始的地址”，开发者应该写为“192.168.0.1/24”。

因此，例如，在Python中，如果开发者受到来自包含192.168.72.12的IP网络区域的DoS攻击，那么，应该在`dos.yaml`中加入以下内容：

```
blacklist:
- subnet: 192.168.72.12/22
```

我说过，DoS保护服务是App Engine的一个新增功能，所以当读者阅读到这里的时候，该服务可能已经有所变化。但是，基本概念是相同的：开发者定义谁需要被阻塞，并将其放到开发者的应用程序的配置文件中。至于有关详情，确实需要查看App Engine网站上最新的文档。

现在，我们知道了云应用程序可能会面临的问题。在本节中，我们已经看到了开发者需要考虑的威胁——网络上的恶意攻击者可能攻击开发者的服务的方式，以及开发者可以如何保护自己免受这些攻击。

17.5　参考文献和资源

- “CIDR符号”，http://en.wikipedia.org/wiki/CIDR_notation。
  Wikipedia上的一篇有关标准IP地址符号的文章，用于管理IP地址块，该内容被App Engine的拒绝服务保护系统所使用。
- OpenSSL，http://www.openssl.org/。
  一个标准的、被广泛使用的安全套接字层的实现。该网站包括了SSL库的漂亮的实现以及丰富的文档，还有底层SSL协议，还介绍了如何正确地使用它。
- 通用网关接口（CGI），http://www.w3.org/CGI/。
  CGI的官方标准和文档。
- “理解拒绝服务攻击”，http://www.us-cert.gov/cas/tips/ST04-015.html。

一篇来自美国计算机应急准备组的优秀文章，解释各种拒绝服务攻击的机制，以及如何识别、抵御和应对这些攻击。

17.4　小结

在这一章中，我们已经很浅显地了解了一下安全性。安全性是一个重要的、复杂的话题，超过了所有构建网络服务的其他方面。正如所看到的，从概念上建立一个安全的应用程序或服务并不困难，但是，实际开发中这几乎是不可能的。

任何安全系统的起点都是安全策略——描述开发者的应用程序/服务管理的对象，以及谁允许访问/管理每个对象的规则。我们已经看到了一个聊天应用程序的安全策略的例子，这个例子是根据聊天室和用户定义的。仔细设计安全策略很关键，我们看到了一个例子，提供一个不使用安全策略的视图可以完全否定这个安全策略。

开发者有了一个设计良好的安全策略后，就需要实现它。认真做到这一点很关键。开发者需要考虑用户可能与应用系统交互的所有方面，并确保系统能够仔细地保护数据，并且总是验证用户执行的是安全策略允许的特定操作。开发者需要坚持这样，应用程序需要检查、检查、再检查用户的证书/身份验证。

开发者设计其系统时需要有这样的想法：它肯定会受到攻击。开发者需要意识到所开发的系统可能受到的不同类型的攻击，实现防止这些攻击的安全措施，监视应用程序，检测主动攻击，如DoS，并且，当检测到攻击时，安排好相应的工具进行应对。

第 18 章

管理App Engine部署

到目前为止，我们几乎完全侧重于如何编写Google App Engine应用程序。这是因为使用App Engine工作中最难的部分就是实现应用程序。当开发者已经能够使应用程序工作之后，管理应用程序的确很容易。

开发者确实需要管理自己的App Engine服务和应用程序。虽然这很容易，但需要持续进行。本章将快速介绍一下App Engine控制面板提供的可用工具，将介绍开发者如何监视其使用的资源量，如何扫描自己的应用程序中存储的数据以找出问题，以及如何看到谁在哪里使用开发者的应用程序等各种问题。

18.1 监视

监视开发者的应用程序只是指查看应用程序是如何被使用的，以及应用程序使用了哪类资源。在App Engine中，查看开发者的应用程序到底发生了什么确实很容易。只要开发者访问http://appengine.google.com并登录，就会得到一个其已经部署的应用程序列表。通过点击应用程序的名字，开发者可以查看其中任何一个应用程序。

当开发者点击一个应用程序时，会被带到该应用程序的主App Engine控制面板，有一个向开发者显示其应用程序的仪表板（dashboard）的视图。仪表板展现开发者可能关心的数据的简要概述。我们可以在图18-1中看到我的Java聊天应用程序的App Engine控制面板的一个例子。正如所示，虽然该程序看起来没有被大量使用，但是，使用信息都在控制面板中。

在顶部，开发者会看到一个图表，显示该应用程序每秒处理的平均请求数量。对于我的Java聊天程序而言，可以看到，使用量很少，其平均使用率是零！真正连续使用一段时间后，我能看到，过去一段时间的使用负载略微提升，达到约每秒两个查询。

顶部的图表可以向开发者显示很多不同内容的摘要。在它的左上方，有一个下拉菜单，开发者可以选择想要查看的视图。除了每秒的请求数之外，还可以看到用于处理每个请求的时间的图、错误的数量、每个请求的带宽、每秒的CPU使用量、以及配额拒绝的数量（即，因为开发者使用完所有的App Engine资源而被拒绝的请求）。例如，在图18-2中，可以看到每个请求花费了多少时间的视图。根据该视图，当我使用聊天软件时，应用程序平均每个请求所用的时间约在24毫秒左右。

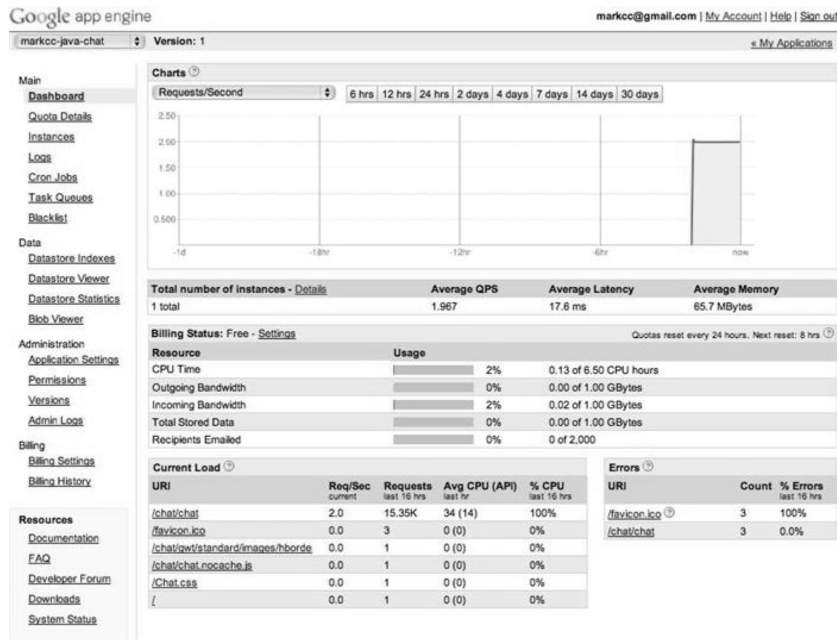


图18-1　JAVA聊天应用程序的控制面板



图18-2　每个请求的CPU时间视图

在该图的下方，有一个更详细的数据集合。

- **实例**

  每个正在运行开发者的处理程序的进程称为一个实例。该图下方有一条线，告诉开发者其应用程序有多少个实例，处理查询的速度是多快，以及使用了多少内存。我的Java聊天程序一般运行一两个实例，主要是由App Engine云中的随机因素决定的，和负载无关。（一般情况下，开发者所得到的实例的数量与开发者的应用程序的负载成正比。在这个例子中，负载很小，一个或者两个实例实际上是随机的。）我的应用程序平均每秒大约两个请求，所以有一个实例时，它显示的平均每秒查询约为2；有两个实例时，它们每个平均大约每秒1个查询。

应用程序实例的平均延迟时间——也就是说，从它们收到消息，到可以开始处理消息时真正花费的时间——只有17毫秒。但是，这些实例每个使用65兆内存！从字面上看，这听起来很疯狂，但请记住，我们正在看一个Java应用程序，所以这个数字包括整个JVM和它加载的所有类库。在内存方面，Java并不算高效。我的Python聊天室的平均值约是其五分之一。但是，考虑到App Engine提供给我的资源的数量，那确实不是一个大问题——应该不会影响开发者选择要使用哪种语言。

- 账单状态

  账单状态告诉开发者其是否为App Engine付费。如果是，那么会向开发者显示已经付费的资源列表，以及这些资源中开发者已经使用了多少。默认情况下，它只标明"免费"，因为如果开发者使用的是随帐户供给的默认资源，这些资源就是免费的。

  如果开发者发现用尽了免费状态的资源，那么，在账单状态部分有一个链接，开发者可以在此购买更多的资源。我们将在18.5节中讨论资源购买的内容。

- 资源

  这是一个开发者可以使用的资源列表，以及一些显示每个资源已经使用了多少的小图表。它列出了CPU时间、下行带宽、上行带宽、存储的数据，以及电子邮件的收件人。例如，在我的聊天室应用程序中，因为程序没有被大量使用，所以，今天在可用的6.5小时CPU时间中，已经用了0.14小时，使用了大约可用带宽的3%。这主要是由于我一直让客户端打开着，并在过去的三小时中每秒运行两个AJAX调用所造成的！

从这些信息中，已经可以得出一些有趣的结论。查看带宽使用率时，我可以看到一个问题。有一个客户端在访问我的聊天室，但我实际上没与任何人聊天。其实是应用程序每秒稳定地运行两个查询，稳步消耗带宽。如果真的有十几人挂在上面，并聊天几个小时，可能真会耗费完带宽配额。因此，如果我真的想使用此应用程序，就需要减少AJAX更新的频率，减少更新请求的大小，或者购买更多的带宽。对于该应用程序，最好的选择可能是同时采用两种方法。我并不真的需要每秒更新一次以上，这样就可以仅通过减少AJAX更新的频率减少一半的带宽使用。除此之外，请求确实是非常小的，而且我能够做的减小请求的工作不多。所以，我需要购买更多的带宽。

在仪表板上，如果默认的视图不能给开发者提供足够的信息，那么，开发者可以深入查看，以获得更多的细节。几乎每个内容都包含链接，通过连接可以让开发者获得更多的信息。例如，可以通过点击邻近"实例的数量"（number of instances）的"详情"（Details），获得更多有关实例的信息。

此外，让我们看一下面板的左侧，有一个不同类别的列表，开发者可以进一步深入查看，得到更多精确的信息。开发者可以检查配额的改变情况，可以得到实例的非常精确的信息，可以检查自己的任务队列，等等。

18.2　小探数据仓库

正如我们在整本书中看到的，App Engine的大部分内容并不是什么新东西。在可能的情况下，

App Engine尝试重用各种标准的技术支持开发者的云应用程序编写。App Engine真正的不同之处，也就是真正新颖的一个内容，就是数据仓库。大多数云应用程序框架向开发者提供对一些轻量级关系数据库的访问，如MySQL。但App Engine却走了一条极为不同的路线，使用数据仓库将内容存放在Google的Bigtable中。

因此，当开发者管理其App Engine应用程序时，真正重要的事情就是理解数据是如何存放的，以及该存储模型会如何影响资源使用。

在App Engine控制面板中，有一组视图可以让开发者窥视和探测其应用程序是如何使用数据仓库的。

最容易查看的内容是开发者的数据索引方式。开发者只要点击“数据仓库索引”（Datastore Indexes），控制面板就会显示数据仓库中存储的数据类型列表，以及这些数据类型的哪些字段被索引。这在早期的开发过程中非常宝贵。数据仓库基于开发者的查询自动索引一些字段。通过查看这个视图，开发者可以看到有什么索引，以及没有什么索引。

开发者也可以用自己的方式浏览其存储的所有对象。开发者只要点击“数据仓库视图”（Datastore Viewer），就可以看到在数据仓库中的所有对象的列表，按类型分类。开发者还可以查找特定的对象，只要点击数据仓库视图中的“选项”（Options）链接，它就向开发者显示一个用于生成当前视图的GQL查询，而且开发者可以编辑该查询。例如，在我的聊天应用程序的数据仓库中，有`PChatMessage`对象和`PChatRoom`对象。我可以输入如`SELECT * FROM PChatMessage where senderName='markcc'`这样的查询，看到我所发送的所有聊天消息。

最后，开发者可以看到对象占用了多少存储空间——包括它们的索引的元数据——在“数据仓库统计”（Datastore Statistics）下查看。

18.3　日志和调试

当应用程序有问题时，整个控制面板上最有用的东西就是日志。日志按照与开发者使用本地调试器差不多的方式，提供了一种检查所运行应用程序的方法。一旦应用程序中任何内容发生错误时，App Engine就会生成一个日志项，通过查看日志，开发者就可以看到发生了什么错误。

例如，打开日志视图时，就可以看到我在聊天应用程序中犯了一个错误。用于生成聊天视图的模板报告说应该有一个网站图标，但并没有找到可用的网站图标。因此，每个页面浏览请求结束时，会由于缺少网站图标产生一个错误。这并不会导致应用程序无法工作。事实上，在第一次检查日志之前，我甚至不知道犯了这样的错误！但这正是日志为何如此有价值的原因。在App Engine中，开发者不能像在自己的计算机上一样直接观察到应用程序的运行。应用程序运行在其他地方的一台机器上，开发者不能直接访问。但经由日志这种宝贵工具，开发者就可以尽可能多地了解正在运行的应用程序的信息。

当然，App Engine并不能自动告诉开发者全部事情。它无法准确地知道开发者想要哪些信息。因此，它自动做的全部工作就是记录下来它所知道的错误。如果应用程序试图从数据仓库访问无法找到的数据，App Engine就会知道该事件，就可以在日志中生成一个对应的表项。如果应用程

序成功地从数据仓库中取回了信息，但却由于查询语句的错误而导致该信息不正确，那么，由于App Engine无法知道信息是否正确，所以它根本不会自动记录任何内容。

可以使用编程语言中的标准日志支持（如Java中的`java.util.logging`）在应用程序中插入日志命令。所有日志内容通过点击控制面板中的“日志”（Logs）选项进行查看，这是开发者最宝贵的调试工具。开发者可以通过生成日志项，准确地看到其应用程序在做什么。

例如，现在部署好的聊天应用程序的版本包含了一个错误。如果给应用程序一个长聊天消息，它将无法正确地处理。为什么呢？

在查看缺省的日志时，我看到一堆条目，像图18-3中所示，在看到的页面上，都是抱怨不能处理长于500字符的聊天消息，因为数据仓库模型将消息内容声明为字符串特性，而字符串不能长于500字节。但我发出的聊天消息没有500个字符长度！

```
E 11-28 01:24PM 32.444
javax.servlet.ServletContext log: Exception while dispatching incoming RPC call
com.google.gwt.user.server.rpc.UnexpectedException: Service method 'public abstract void
com.pragprog.aebook.chat.client.ChatService.postMessage(com.pragprog.aebook.chat.client.ChatMessa
ge)' threw an unexpected exception: java.lang.IllegalArgumentException: message: String
properties must be 500 characters or less.  Instead, use com.google.appengine.api.datastore.Text,
which can store strings of any length.
```

图18-3　控制面板日志视图的一个日志项

对于这个问题，默认的日志信息足以提供一定的线索，但我们先假设它没有提供充足的信息。也许我会认为聊天消息并没有那么长，那么可以在应用程序中添加日志代码跟踪调试。首先，需要在源文件中声明日志记录器。因此，我在`com.pragprog.aebook.persistchat.server.ChatSubmissionServiceImpl.java`中添加了如下内容，在最后面，右括号之前：

```
private static java.util.logging.Logger logger =
    java.util.logging.Logger.getLogger(
            ChatSubmissionServiceImpl.class.getName());
```

然后，在`PostMessage`方法里，添加一个日志语句：

```
logger.info("Chat message size = " + message.getMessage().length() +
        " with body \"" + message.getMessage() + "\"");
```

通过添加的语句，每一个发布的消息都会产生一个日志条目，包含发布消息的长度和内容。就这样，我可以确认消息确实超过了500个字符。

不过，给开发者的应用程序添加日志时应该仔细一些。虽然这是一个非常优秀的工具，但日志却会占用CPU时间，会算作开发者的配额。（开发者并不需要担心空间，App Engine只保留一定数量的最新的日志条目，自动丢弃旧的条目）。因此，日志太多的话，渐渐地就会浪费资金。虽然不多，但却是一个现实问题。

更重要的是，日志是开发者理解应用程序实际状况的主要工具。如果为每件小事都生成一大堆的日志条目，那么日志里就挤满了不相关的数据。这些数据会使得重要的条目很难找到，或者如果存在很多不相关数据，甚至可以完全挤出有用的数据！依照经验来看，总的原则是不记录你不想主动阅读的内容，不要添加那么明知道永远都不会看的日志条目。但当遇到问题时，就可添

加日志语句，以便了解发生的问题，然后进行调试时，点击打开控制面板中的日志即可。

18.4 管理应用程序

控制面板包含了一组设置，可以帮助开发者管理其应用程序。有4个视图可以用来管理：应用程序设置（Application Settings）、权限（Permissions）、版本（Versions）以及管理日志（Admin Logs）。

在“应用程序设置”视图中，开发者可以访问其应用程序的一组基本设置。包括以下内容。

- **应用程序标题**（Application Title）
  这个可编辑文本框中产生的是应用程序名称。想要改变应用程序名称，只要修改文本框内容并单击“保存设置”（Save Settings）即可。
- **配置的服务**（Configured Services）
  如果应用程序要使用任何服务，如电子邮件或者XMPP（聊天），服务就会在这里列出，还有各个可配置选项。
- **Cookie到期时间**（Cookie Expiration）
  App Engine使用Cookie对用户进行身份验证。开发者可以在设置cookies时选择该用户保持登录一天，还是一个星期。
- **禁用或删除应用程序**（Disable or Delete Application）
  如果开发者想暂时或永久关闭其应用程序，那么，这里可以完成这项工作。只要开发者点击“禁用应用程序”（Disable Application），首先会先得到一个确认对话框，以确保真的要关闭应用程序。如果开发者确认，应用程序就会被禁用，并且Application Settings（应用程序设置）视图也会被更改，变为包含两个选项——一个重新启动应用程序，一个永久删除应用程序。
- **域设置**（Domain Setup）
  如果开发者想在自己的域中运行其应用程序，可以在这里告诉App Engine有关域的信息。目前，开发者可以使用Google App购买并设立一个域，然后，让其App Engine程序运行在该域中。此外，如果开发者拥有的是一个并非由Google App运行的域，那么，就不能将它用于其App Engine服务。

开发者通过权限视图给其他用户管理权限。如果授予另一个用户管理权限，那么他将能够访问开发者的应用程序的整个控制面板，包括可以更改任何设置、修改代码，以及禁用或者甚至删除应用程序。

版本视图向开发者展示了已经部署的应用程序的版本的列表，并且，为开发者提供了选择其中任何一个作为当前活跃版本的能力。这样做的主要用途是防止错误。如果开发者部署了一个新版本的应用程序，但它包含一个错误，那么，他可以通过访问版本视图，选择之前一个没有错误的版本，从而立即恢复到以前的版本。

最后，管理日志让开发者看到其应用程序上执行了什么样的管理操作。这个视图的主要用途

是让开发者监视其他管理员的操作:任何管理员执行的任何操作都会在管理日志中生成一个条目。

18.5　支付用户所使用的资源

本书的开头说过，云计算的基本特点就是让开发者购买想要使用的资源。开发者不用购买整台计算机以确保有足够的CPU应付偶尔到来的高峰，那样只会让那些CPU在大部分时间里都处于闲置状态。开发者只需购买其所需要的CPU即可。帐单面板就是App Engine让开发者告诉Google其想要购买多少资源的地方。

控制面板上有一个“帐单设置”（Billing Setting）的链接。如果开发者点击该链接，它就会开启账单选项。默认情况下，开发者使用免费资源工作——Google给开发者提供首次资源的免费使用——但如果想要比最低资源更多的资源，就需要付费了。

开发者使用帐单进行控制，可以设置一个每日预算。可以设定日最高值——在某一天里愿意为额外资源支付的金额上限。开发者也可以设置每个资源的最大值，比如说:“我愿意每天为CPU花费最多4美元，但带宽只有1.5美元。”

开发者设置了预算后，可以使用Google Checkout在自己的信用卡上设置一个支付项。

通过查看帐单记录，开发者可以精确地看到自己正在使用何种资源，以及每天支付多少钱。在帐单历史记录中有每日的条目，采用相同的格式作为日志项，告诉开发者每天各类资源的消耗是多少。

在这一章中，我们已经快速了解了App Engine的控制面板为开发者的应用程序提供的管理控制。这里的内容显得有些平淡无奇，因为这一切都非常简单，并且易于使用。

第 19 章

结束语

19

在之前的讲述中，我们已经讨论了很多方面的内容。此时，开发者已经熟悉Google App Engine编程的基础知识了。当然，我们无法涵盖所有内容，但是，现在开发者已经可以构建自己的第一个应用程序，并使用Google App Engine的在线文档工作了。在本章中，我们将回顾一下已经介绍的基本概念，并介绍一下开发者可能需要看的其他内容，以及后续如何进一步发展。

19.1 云的概念

正如我们已经看到的，云计算编程与在传统环境中的编程非常不同。使云这么有趣和强大的关键特性包括如下内容。

- 开发者不知道——或者说是不关心——其程序在哪里运行。这是云的最根本的特征之一。开发者不是在一台计算机上运行应用程序：云服务提供商为开发者提供了一个平台，更确切地说，是一个可以运行以特定方式编写的程序的基本软件系统，而且所有的底层细节是平台的问题，而不是开发者的问题。通常，开发者不知道应用程序运行在什么样的电脑上，或者运行在多少台电脑上，这些计算机在哪里，或者运行什么操作系统。这些都无关紧要。
- 软件是一种服务。在传统的编程世界里，开发者实现运行于自己的计算机上的应用程序。在云中，开发者仍然编写应用程序，但是从客户的角度来看，云中的程序与传统的程序并不是完全相同的，因为客户不运行该程序。在云中，开发者的软件是一种服务，是一个运行在其他地方的应用程序，当用户想要使用该程序时，通过他们的网络浏览器进行访问。关键的区别是，用户不运行该程序，程序总是处于运行状态。当用户想要使用该程序时可以连接到它，而且用户并不需要担心如何运行程序，程序是否需要升级，程序是否在他们的计算机上能正常工作，或者类似的事情。服务是永远存在的，随时可以被任何地方的任何计算机访问。
- 没有任何限制。开发者不必担心类似“我的计算机可以处理多少个页面视图”这样的限制和问题。在云中，一切都是有弹性的。如果开发者突然有了更多的用户，也不是问题：云平台只是按照用户需要获取更多的资源，并继续向开发者的客户提供服务。
- 开发者为自己所使用的东西付费。在云中，开发者按照规定为自己所使用的各种资源付费。

开发者不需要为了以后可能的需求而购买庞大的、功能强大、价格昂贵的计算机。如果今天只需要一个小规模处理器，而明天需要强大一千倍的处理器，那也没问题。今天，开发者为一个处理器支付几美分，明天，可能为某些强大的计算机支付几美元。无论开发者需要什么，都可以在需要时为其付费，而且使用完后，不必继续维持过度的硬件资源。

19.2 Google App Engine 的概念

正如本书介绍的，Google App Engine是围绕着一系列基本概念构建的。回顾一下，使用Google App Engine进行云编程的基本概念包括以下内容。

- 编程的过程就是实现HTTP请求处理的过程。在Google App Engine中，开发者的编程工作就是处理HTTP请求。通过神奇的网络回调函数，开发者的程序与用户以及其他应用程序的每一次交互，都可以实现成HTTP请求处理程序。
- 开发者的工作基础是各种标准。在Google App Engine中，如果可能的话，每个功能尽可能基于已有的标准技术实现。因此，例如在Python中，模板是采用被广泛使用的Django模板框架构建的。在Java中，数据仓库的接口是按照标准的Java数据对象框架构建的。纵观Google App Engine，我们已经看到，在可能的情况下，它会采用已有的标准技术作为其基础，而不是发明新的东西。
- 一切都是服务。开发者在其程序中进行交互的Google App Engine的所有部分都是服务。与在传统的程序中链接的函数库的情况类似，在Google App Engine的云应用程序中，开发者是通过RPC请求来与服务进行交互的。而且，它有大量服务可以被开发者使用，提供了安全、数据存储、即时消息等全部内容。
- 浏览器是用户界面。用户界面的每一部分，用户所看到的一切内容，用户所做的一切事情，都发生在网络浏览器里。
- 没有状态。在传统的程序里，开发者可以依靠变量来保存状态——也就是说，如果开发者把某内容存放在一个变量里，稍后回来查看该变量时，它仍然存在。在云环境中，这不一定正确。开发者编程时必须意识到：保持状态的唯一途径就是使用数据仓库中的外部存储。
- 时刻牢记安全。在云应用程序中，安全比在传统的应用程序中更需要技巧。由于开发者的代码作为服务在运行，可以被恶意攻击者在线访问，而且用户的数据都可以被在线服务访问，因此，开发者必须格外注意确保系统的构建是安全的。从设计系统的第一步开始，就要开始设计安全策略，并确保在设计的每一个细节坚持执行此安全策略。所有的错误都将危及用户数据的安全。

这些基本思想组成了Google App Engine的核心。虽然没有很多内容，但这就是重点。正是基本的简单性，才使App Engine成为用起来如此完美、功能强大的系统。Google App Engine中只有这组适度的概念和围绕这些概念构建的库，仅此而已。但是，这种简单的框架成为了开发者用于构建自己的云应用程序的非凡的、一流的工具。

19.3　路在何方

Google App Engine是一个非常新的平台，它在不断地发展和成长。在我写这本书的时间里，很多内容改变了，以至于我不得不重写相关的整个章节。当读者读到这里的时候，Google App Engine里可能会有更多新的东西。如下是一些即将出现的令人兴奋的变化。

- 数据仓库选项。在写这本书的时候，只有一个数据仓库的实现。现在，在我写这个总结的时候，第二个数据仓库正在测试中。开发者可以在两个有相同接口的不同实现中进行选择，从而能够选择最适合其应用程序的实现。当前的实现称为主-从数据仓库（master-slave datastore）。主-从数据仓库仍然会是默认的版本。但是，如果采用主-从数据仓库，那么，在数据中心的维护窗口期间，开发者数据的访问有可能会被中断。另一种是高复制数据仓库（high-replication datastore）。高复制数据仓库速度稍慢（最多两倍），并具有最终一致性，但是，可以保证开发者的数据即使面对重大破坏，也始终可用。
- 开发存储。这是当前测试部署的令人兴奋的开发之一。Google提供了非常类似Amazon S3的存储服务。事实上，开发者可以很容易地将为S3编写的程序移植为使用Google开发存储。Google开发存储也有一些自身的API，使其成为一个非常有吸引力的平台。这是App Engine中的一件大事。
- MySQL数据库支持。如果开发者已经有一个应用程序，并且想要将其迁移到云或者开发者真正需要或者喜欢的关系数据库上，那么，Google App Engine团队实现了一个MySQL版本，已将其扩展到在App Engine云内工作。有了这个数据库，开发者就可以使用MySQL关系数据库存储来代替数据仓库了。
- Go语言支持。Google的一个团队开发了一个漂亮的新的系统编程语言，名为Go。我认为，Go最终会被作为开发Google App Engine程序的语言。Go是一个优秀的语言，它结合了Python的简单性和简洁性以及Java的安全性和类型安全。当Go语言出现时，在我看来，它会是Google App Engine开发语言的首选。
- Django nonrel。目前，Google App Engine的Python开发使用的是Django框架的模板库。Django的其他内容，特别是其数据管理框架，非常优秀，但是，与数据仓库管理数据的方式不兼容。现在正在努力构建一个Django的变种版本，希望其能够与数据仓库类似的非关系对象存储一起正常工作。如果这项工作完成，在GAE开发中，Django将被作为网络应用的一种功能齐全的替代方法。
- 通道。目前在Google App Engine中，客户需要从App Engine的服务器请求数据。在聊天应用程序中，我们看到，使用这种机制，要每两秒就发送更新请求，从而保持聊天窗口是最新的。通道是目前正在推动支持的一种机制，它提供了一种方式，使得运行在云服务器上的Google App Engine服务能够向客户端用户界面发送更新，而无需任何请求。有了通道，不使用轮询更新就可以实现我们的聊天服务：服务器只要使用一个通道将更新推送到客户端就可以。

- MapReduce。对于数据仓库中的数据的复杂计算，目前开发者只能使用任务队列来进行简单处理。Google App Engine团队正在开发对Google MapReduce框架的支持，该框架用于大规模并行计算。当这项工作完成时，Google App Engine将可以被用于各种实现并行计算的工作负载，如生物信息学处理等。
- 更好的SSL支持。现在，只有当开发者构建运行于appspot.com域的应用程序时，Google App Engine才允许开发者使用SSL安全通信。不久，开发者就可以通过将SSL证书提供给Google App Engine，在自己的域中使用自己的SSL密钥了。

这些仅仅是一些大概，还有非常多的内容。读者应该跟进App Engine博客中关于所有变化的信息——包括我上面提到的，以及更多的信息。

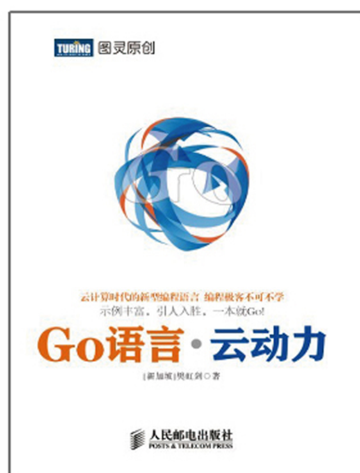
正如我们看到的，Google App Engine是一个巨大的用于构建基于云的应用程序的系统。现在，你已经知道了如何使用它，所以，出发吧，去构建一些伟大的应用！

19.4 参考文献和资源

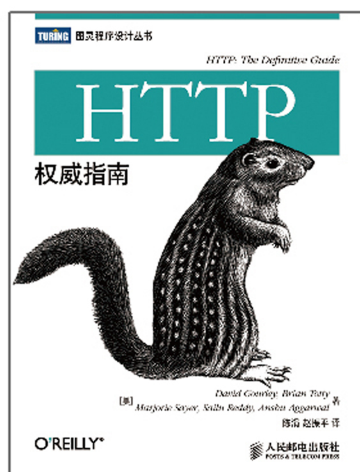
- Google App Engine博客，http://googleappengine.blogspot.com/。
  官方的Google App Engine博客。每一个App Engine开发人员都应该跟进此博客。
- Google App Engine的社区主页，http://code.google.com/appengine/community.html。
  App Engine社区的Google代码交换场所。这个网站包括了论坛，开发者在那里可以得到各种问题的回答、示例应用程序的下载，讨论即将推出的功能、App Engine的系统状态信息和常见问题等，这是一个对Google App Engine开发者来讲非常棒的资源。



云计算
核心技术剖析
吴朱华　编著
一线云计算专家倾情奉献
深入分析多种云计算核心技术
通过实例教你编写简化版的BigTable
人民邮电出版社



TURING 图灵原创
云计算时代的新型编程语言 编程极客不可不学
示例丰富，引人入胜，一本就Go!
Go语言·云动力
[新加坡]樊虹剑○著
人民邮电出版社
POSTS & TELECOM PRESS



TURING 图灵程序设计丛书
HTTP
权威指南
O'REILLY
人民邮电出版社
POSTS & TELECOM PRESS

“如果你想了解云技术并学习Google App Engine的话，本书绝对是上上之选。你可以看到用两种语言（Python和Java）编写的范例。在介绍技术方面，Mark做得实在很出色。他用词准确，从不故弄玄虚，阅读体验极佳，让我沉浸其中。”

——Fred Daoud，*Stripes：…and Java Web Development Is Fun Again*作者

“简洁的代码注释详细，优美的行文解释清晰，这本书可以满足云技术开发初学者的全部需要。”

——Dorothea Salo，美国威斯康星大学麦迪逊分校

Code in the Cloud

Programming Google AppEngine

云端代码 Google App Engine编程指南

云计算彻底改变了应用程序的开发与使用方式，甚至也改变了应用程序原本的定义。有了云计算，应用不再运行在用户桌面计算机上，而是分布式地运行于网络上，与全世界千万用户同时使用计算资源。它还具有传统应用程序所不可比拟的功能多样性及可扩展性。在诸多构建云服务的新环境中，Google App Engine以其强大的功能和易用性无疑成为非常吸引人的一个框架。

本书阐述了云应用的内涵，剖析了其与传统应用的区别，并通过使用Python与Java对一个简单的应用进行不断的深入开发，揭示出App Engine的各方面特性，从而使读者顺利掌握构建云端应用程序的秘诀。

本书主要内容

- 云服务的内涵及其与传统应用程序的区别
- 如何应用Python或Java，采用迭代方式开发简单的云应用程序
- 如何利用App Engine提供的便利服务
- 如何建立在用户浏览器上运行的交互式用户界面
- 如何构建云服务
- 如何使用App Engine管理持久性数据
- 如何保障Web应用程序的安全性
- 如何与运行在App Engine云端的其他服务进行交互



图灵社区：www.ituring.com.cn
新浪微博：@图灵教育 @图灵社区
反馈/投稿/推荐邮箱：contact@turingbook.com
热线：(010)51095186转604

分类建议 计算机/程序设计/云计算

人民邮电出版社网址：www.ptpress.com.cn

The Pragmatic Programmers



ISBN 978-7-115-30199-4
定价：45.00元